OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes

Pierre Crescenzo

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR SCIENCES

École Doctorale
Sciences et Technologies de l'Information et de la Communication

THÈSE

présentée pour obtenir le titre de

Docteur en Informatique

de l'Université de Nice-Sophia Antipolis

par

Pierre Crescenzo

OFL : UN MODÈLE POUR PARAMÉTRER LA SÉMANTIQUE OPÉRATIONNELLE DES LANGAGES À OBJETS

APPLICATION AUX RELATIONS INTER-CLASSES

soutenue publiquement le 20 décembre 2001 devant le jury composé de :

Président :	**Jean-Paul Rigault**	Université de Nice-Sophia Antipolis
Rapporteurs :	**Christophe Dony**	Université de Montpellier II
	Mourad Chabane Oussalah	Université de Nantes
Directeur :	**Robert Chignoli**	Université de Nice-Sophia Antipolis
Examinateurs :	**Philippe Lahire**	Université de Nice-Sophia Antipolis
	Michel Riveill	Université de Nice-Sophia Antipolis

à 9 h 30 à Sophia-Antipolis

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR SCIENCES

École Doctorale
Sciences et Technologies de l'Information et de la Communication

THÈSE

présentée pour obtenir le titre de

Docteur en Informatique

de l'Université de Nice-Sophia Antipolis

par

Pierre Crescenzo

OFL : UN MODÈLE POUR PARAMÉTRER LA SÉMANTIQUE OPÉRATIONNELLE DES LANGAGES À OBJETS

APPLICATION AUX RELATIONS INTER-CLASSES

soutenue publiquement le 20 décembre 2001 devant le jury composé de :

Président :	**Jean-Paul Rigault**	Université de Nice-Sophia Antipolis
Rapporteurs :	**Christophe Dony**	Université de Montpellier II
	Mourad Chabane Oussalah	Université de Nantes
Directeur :	**Robert Chignoli**	Université de Nice-Sophia Antipolis
Examinateurs :	**Philippe Lahire**	Université de Nice-Sophia Antipolis
	Michel Riveill	Université de Nice-Sophia Antipolis

à 9 h 30 à Sophia-Antipolis

Table des matières

Table des figures

Liste des tableaux

Index

Programmez toujours vos applications comme si la personne qui doit les maintenir ensuite était un psychopathe violent qui sait où vous habitez.
John F. Woods

Remerciements

Je tiens tout d'abord à remercier mon Directeur de Thèse, Robert Chignoli, d'avoir orienté mes travaux tout au long de mon étude doctorale et d'avoir été présent à chaque fois que le besoin s'en faisait sentir. J'adresse également mes remerciements à Philippe Lahire qui a co-encadré cette thèse et dont la présence constante et l'attention sans faille, je veux même dire l'amitié, m'ont permis de mener à bien cette étude.

Je remercie tous les membres du Projet *Objets et Composants Logiciels* du Laboratoire *Informatique, Signaux et Systèmes de Sophia-Antipolis* de m'avoir accueilli et soutenu. Je tiens plus précisément à exprimer ma gratitude envers Roger Rousseau, Chef du Projet, dont l'expérience et les conseils me sont précieux. Je le remercie également de m'avoir donné, quelques années auparavant pendant ses enseignements que je suivais, l'envie de poursuivre dans cette voie. Je remercie Philippe Collet de ses conseils, pragmatiques et efficaces, et de sa camaraderie sincère. Je remercie enfin Adeline Capouillez de son assistance et surtout de me supporter avec tant de patience au sein du même bureau.

Je tiens également à adresser mes remerciements aux membres de mon jury, qui m'ont fait l'honneur d'accepter d'évaluer mon travail : Jean-Paul Rigault, Christophe Dony, Mourad Chabane Oussalah et Michel Riveill.

Je souhaite aussi à remercier les autres membres du laboratoire pour leur sympathique présence. Je citerais pêle-mêle Cathy, Michel, Daniel, Annie, Vincent, Laurent, Carine, Michel, Cécile, Fabrice, Charles, Patrick, Jean-Pierre, Gilles, Isabelle, Patricia, Robert-Olivier, Olivier, Manuel, Annabelle, Stéphanie, Lionel, Gilles, Lionel, Guy, Mireille, Anne-Marie, Martine, Hélène, Céline, Jacques, Érick, Micheline, Corinne, Patricia, Jean-Claude, Emmanuel, Aziz, Chan, Laurent, Nhan, Olivier, Igor, Bruno, Luc, Marie-Agnès, Christian, Maria-Joao, Pierre, Jean-Paul, Laurent, Manuela, Pascal, Xavier, Stefan, Viviane, Michel, Peter, Jean-Luc, ...

Je remercie évidemment mes amis Emmanuelle, Olivier, Marilyne, Jean-Louis, Marc et Greta de leur présence fidèle et vraie. Les moments passés ensemble furent parmi les meilleurs et le seront encore.

Je ne terminerai pas sans adresser un merci tout personnel à toutes celles qui, tout au long de ces années, m'ont laissé puiser dans leurs yeux et leur sourire l'énergie de poursuivre ce travail. Et encore et bien sûr merci à toi, Florence, compagne attentive, tolérante et passionnée d'avoir, avec tant d'élan, partagé avec moi les plaisirs et difficultés de cette dernière année.

Enfin, je remercie toute ma famille et particulièrement mes parents de m'avoir toujours appuyé et soutenu, surtout dans les moments les plus pénibles. Cette thèse, modeste résultat en regard de leur dévouement, leur est dédiée.

Chapitre 1

Introduction

Améliorer la qualité des logiciels. C'est le désir légitime de toute personne qui s'intéresse à la programmation informatique en général et au génie logiciel en particulier. Mais, derrière ce but évident se cache apparemment de nombreuses contradictions. Certains considèrent que le critère déterminant est la performance, que le logiciel doit donc être le plus rapide possible. D'autres mettent plutôt l'accent sur la fiabilité c'est-à-dire sur la capacité du logiciel à donner des réponses exactes et vérifiées dans une situation donnée. D'autres encore privilégient la robustesse, la capacité du logiciel à continuer à fonctionner dans un contexte matériel ou logiciel dégradé. Et ce ne sont là que quelques exemples... L'amélioration de la qualité globale relève donc soit d'un choix qui avantage une qualité au détriment des autres, soit d'un compromis entre ces qualités.

Vient ensuite la question des moyens. Diverses pistes sont possibles pour concrétiser l'amélioration de la qualité des logiciels. La progression des technologies des matériels apporte un outil utile. L'évolution des environnements de programmation, des compilateurs, interprètes, machines virtuelles, etc. sur lesquels reposent les logiciels est un autre support efficace. Une troisième voie intéressante consiste à privilégier la qualité du code source des applications en tablant sur les conséquences naturelles qui en découleront lors de l'exécution : assainir les fondations pour améliorer l'ouvrage. C'est cette dernière voie que nous avons décidé de suivre dans l'étude qui vous est présentée ici.

1.1 Contexte et objectifs

Notre but principal est de fournir au programmeur des outils lui permettant d'améliorer la qualité du code source de ses applications. Le contexte dans lequel nous nous plaçons est celui des langages à objets à classes, tels *Eiffel*, *Java*, *C++* et *Smalltalk*. Les langages qui nous intéressent sont donc souvent typés[1] et impératifs, bien que cette dernière caractéristique soit en fait secondaire.

Ce travail se place également dans une optique de génie logiciel. Comme tous les domaines inhérents à l'ingénierie, le génie logiciel se nourrit principalement de *pragmatisme*. Notre idée du génie logiciel est en effet qu'il doit guider le programmeur[2]

[1] Ils sont même pour la plupart qualifiés généralement de *fortement typés* car ils effectuent d'importants contrôles de cohérence du typage.

[2] Le génie logiciel guide bien d'autres intervenants que le programmeur mais c'est cette phase-là qui nous intéresse ici.

pour améliorer son travail aux moyens de méthodes et techniques. Un exemple éclairera, nous le pensons, le lecteur sur ce fondement de notre approche.

Nous considérons le système d'assertions d'*Eiffel* comme une technique de génie logiciel à l'usage du concepteur de programmes qu'est le programmeur. Son utilisation lui permet d'améliorer la qualité de son code par l'ajout d'informations supplémentaires, les assertions, qui vont être mises à profit pour réaliser des tests, pour contrôler des algorithmes, pour mieux spécifier la documentation, ... Un usage systématique et complet des assertions serait cependant particulièrement pénible. Faut-il pour cela nous en passer ? L'absence d'assertions dans un code source limite la qualité de celui-ci. Faut-il pour cela obliger le programmeur à en mettre de partout ? Nous croyons que le compromis que nous dicte le bon sens est une approche correcte. Les assertions doivent être utilisées à chaque fois que le temps et la volonté du programmeur le permettent ou à chaque fois que le souci de qualité le requiert.

Les assertions sont une des préoccupations du projet OCL (Objets et Composants Logiciels) au sein duquel cette thèse, dirigée par Robert Chignoli, a été menée. Cependant, c'est un autre angle qui nous occupe ici : les *relations entre classes*.

Les relations entre classes des langages à objets courants sont mises en œuvre pour des usages souvent très différents. Par exemple, l'héritage est aussi bien utilisé pour faire de la spécialisation que pour gérer des versions, réutiliser du code voire pour implémenter une classe abstraite. Ce choix est celui de la simplicité immédiate : un seul mécanisme permet de réaliser plusieurs relations sémantiquement proches. En plus de la simplicité, cette option n'oblige pas le programmeur à faire l'effort de formaliser outre mesure sa relation : il sait que l'héritage convient et que ce soit de la spécialisation mâtinée de réutilisation de code ou quelque chose d'approchant lui importe finalement peu.

Les avantages s'accompagnent des inconvénients associés. Ici, nous pouvons par exemple évoquer les points suivants :

- Il est très généralement admis qu'un effort de formalisation en amont fait perdre du temps à court terme en aval mais avantage les perspectives à long terme. Par exemple, un programmeur qui néglige la phase de conception démarre plus rapidement l'implantation de son application mais en ressent rapidement les inconvénients. Nous pensons que si le programmeur prend le temps de définir avec plus de précision ses relations, son graphe de classes s'en trouvera mieux spécifié.
- Le code source doit, autant que possible, mettre en évidence précisément les choix du programmeur. Plus l'expression est généraliste, plus elle est applicable mais moins elle est à même de spécifier exactement le comportement souhaité. Si le programmeur spécifie une réutilisation de code, il ne sera pas nécessaire d'essayer de trouver autre chose — comme une spécialisation — dans sa relation d'où un gain appréciable de travail et de temps. De plus, si la relation est plus précise, elle décrit avec plus d'exactitude la sémantique que le programmeur a souhaité lui associer et clarifie donc le *contrat de services* qu'il a souhaité définir. De nombreux contrôles automatiques plus ciblés peuvent aussi être appliqués. Tout cela augmente sensiblement la lisibilité et la maintenabilité de l'application, problème délicat s'il en est, en informatique.
- Une relation reposant sur un mécanisme générique de bas niveau limite forcément les contrôles qu'elle peut effectuer sur sa validité. Le programmeur qui utilise l'héritage pour effectuer de la réutilisation de code ne dispose pas de moyen simple pour empêcher un polymorphisme qui n'a pourtant là pas de sens.

Par exemple, il est envisageable de réutiliser le code de la méthode âge de la classe Personne — le calcul étant fait à partir de dateNaissance — dans la méthode âge de la classe Automobile — à partir de dateMiseEnCirculation. Sauf à modifier significativement le code de ses classes, il ne peut pas empêcher un autre développeur d'utiliser ce polymorphisme inopportun, il ne peut pas indiquer simplement au compilateur cette restriction qui permettrait pourtant des optimisations notables, pas plus qu'il n'est en mesure de spécifier automatiquement le but de la relation dans la documentation.

– Un autre désagrément associé au nombre relativement limité de type de relations entre classes est la difficulté de traduire dans son langage de programmation favori un graphe de classes décrit avec toutes les capacités d'un modèle de conception tel *UML*. Il est évidemment tout à fait possible de traduire une association *UML* en *Java*, par exemple en faisant usage de patrons de conception (*design patterns*). Mais que reste-t-il du schéma initial dans le graphe global des classes *Java* ? Lorsque la sémantique de la relation est éloignée des relations traditionnelles des langages[3], la projection, avec son lot d'approximations, nous semble légitime. Mais lorsque la sémantique est proche de l'héritage par exemple, il nous paraît frustrant de ne pas pouvoir la préciser.

Afin d'améliorer la lisibilité et la maintenabilité du code produit, il serait donc intéressant de pouvoir spécifier plus explicitement, au niveau du langage de programmation, quelle sorte de relation doit être créée entre les classes. Notre propos n'est pas de systématiquement supplanter l'héritage ou toute autre relation entre classes. Il s'agit plutôt d'en dégager des utilisations courantes et de prévoir un système capable d'indiquer que l'on se trouve dans un cas plutôt que dans un autre. Il est ainsi envisageable d'étendre un langage de programmation existant en lui adjoignant un ou plusieurs nouveaux types de relation. Lorsque le programmeur en ressentira le besoin ou la volonté, il pourra alors préciser sa pensée.

Les travaux préliminaires qui ont mené à cette étude nous ont conduit à trois points importants.

1. Adapter le comportement de l'héritage est notre objectif initial. Mais l'héritage est mis en œuvre de manières différentes dans les divers langages et il n'est pas la seule relation entre classes sur laquelle nous pourrions intervenir. Préciser les usages de l'agrégation[4], par exemple, s'avèrerait également intéressant. Pour aller plus loin, nous pensons que d'autres relations pourraient tirer profit d'une meilleure et plus précise spécification telles les relations entre classes et objets (citons l'instanciation), voire entre objets.
2. Voilà pour nous admise l'idée de la création d'un système d'adaptation des relations. Mais s'arrêter là serait un peu vite oublier qu'une relation lie plusieurs entités, ici principalement des classes. Naturellement, le processus d'adaptation des relations devra donc, selon nous, être généralisé et s'accompagner de capacités similaires pour les classes.
3. Pour présenter un intérêt suffisant et pour ne pas enfermer l'utilisateur dans des choix forcément arbitraires, notre proposition doit être applicable à une famille de langages de programmation plutôt qu'à un seul et les avancées effectuées doivent donc être réutilisables pour plusieurs langages. Cependant, trai-

[3]Par exemple, le programmeur fait usage de l'héritage dans le but de réutiliser du code et sans aucun souci de spécialisation.

[4]Pour simplifier nous pouvons, à ce stade, définir l'agrégation comme la relation mise en place entre une classe déclarant un attribut et celle qui décrit le type de ce dernier.

ter le problème de manière générale à tous langages serait évidemment une tâche trop difficile. Aussi, dans un souci pragmatique, restreignons nous notre approche à un sous-ensemble de ceux-ci.

De ces quelques réflexions est né le sujet de cette thèse : il nous fait définir un *modèle méta-objet*, permettant de décrire et d'adapter le comportement, la *sémantique opérationnelle*, des relations entre classes et des classes dans les langages à objets. Avec toujours à l'esprit le souci de pragmatisme lié au génie logiciel : le surplus de travail requis ne devra pas être disproportionné par rapport aux avantages fournis.

1.2 Plan de ce document

Avant d'entrer dans le vif du sujet, il nous reste à vous présenter le plan que nous suivons au sein de ce document.

À la suite de cette introduction vous trouverez, à partir de la section 2 page 23, l'*état de l'art*. Il propose une visite guidée du contexte de cette étude et de diverses solutions existantes pour répondre à nos objectifs.

Le chapitre 3 page 47, propose une étude des *sortes de classe* et des *sortes de relation entre classes* présentes dans les deux langages qui nous ont inspiré un grand nombre d'idées : *Eiffel* et *Java*. Notre démarche s'inscrit en effet dans une logique de modélisation ascendante, à partir des mécanismes existants, de manière à reposer sur une base solide. Nous avons aussi parallèlement travaillé sur une généralisation et une extension de ces mécanismes de manière à élargir le champ d'application de nos résultats.

Vient ensuite un chapitre essentiel qui décrit le modèle que nous proposons : *OFL*[5]. Cette présentation débute en section 4 page 73 sous le titre *OFL et ses concepts*. Les trois grandes entités adaptables et configurables, en un mot paramétrables — une grande partie de notre démarche repose, comme vous le verrez, sur cette notion de *concept paramétrable* —, du modèle :

1. les *relations*,
2. les *descriptions* qui généralisent la notion de classe et
3. les *langages* eux-mêmes

sont disséquées et leurs capacités d'adaptation expliquées. L'*hyper-généricité*, élément fondamental d'*OFL*, y est décrite.

Dans le chapitre 5 page 103, nous proposons tout d'abord une vision synthétique, reposant sur les concepts déjà expliqués, de l'architecture du modèle *OFL*. Cette vue d'ensemble décrit les trois niveaux de modélisation mis en place. Puis, dans une seconde partie, les *atomes*, réification des éléments de base mis en œuvre par les langages à objets — *méthode* et *instruction* sont deux exemples d'entités ainsi réifiées —, sont détaillés.

La partie structurelle d'*OFL* est alors décrite, il nous faut ensuite proposer la partie opérationnelle du modèle. Cela est fait dans le chapitre 6 page 117 qui présente un système d'*actions*.

Le suivant et avant-dernier chapitre 7 page 151, dégage, en premier lieu, les apports du modèle puis, dans une deuxième section, présente plusieurs exemples de relation entre descriptions illustrant l'usage d'*OFL*. Dans une troisième partie, les principaux outils logiciels concrétisant nos travaux sont décrits.

[5] *Open Flexible Languages*

Enfin, le dernier chapitre 8 page 175, donne les perspectives de poursuite de cette étude et conclut ce document.

Nous vous invitons maintenant à poursuivre la découverte de ce mémoire par la partie concernant l'état de l'art.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre, nous présentons un état de l'art décrivant les différents outils, langages, méthodes et modèles dont nous avons réalisés l'étude. Vous trouverez tout d'abord une description de deux langages à objets à classes : *Eiffel* et *Java*. Ce sont les deux principales sources d'inspiration qui ont motivé cette thèse. En effet, pour garantir la validité de notre approche, il fallait nous assurer que notre démarche pouvait s'appliquer à aux moins deux langages, ce qui ne limite nullement la possibilité de généraliser notre approche à ceux-ci.

Nous regroupons ensuite, à partir de la section 2.3 page 36, les autres sujets sur lesquels s'est porté notre étude :

- les langages méta-objets et les méta-modèles car notre travail consiste à réaliser l'un d'entre eux,
- les modèles et méthodes de conception car nous souhaitons nous attacher à réduire le fossé entre méthodes de conception et langages de programmation, puis
- les patrons de conception qui offrent des capacités intéressantes d'application d'un modèle à un langage donné.

Pour chaque élément présenté, nous indiquons en quoi il peut servir à atteindre ou, tout au moins à approcher, nos objectifs. Nous décrivons également leurs faiblesses en ce domaine.

Enfin, nous faisons un bilan de cet état de l'art, en forme de pré-introduction au modèle *OFL*.

2.2 Langages à objets

2.2.1 *Eiffel*

Eiffel[1] [95, 78, 99, 79] est un langage à objets pur[2] inventé en 1985 par Bertrand Meyer. *Eiffel* fut créé dans l'optique de permettre l'amélioration de la qualité des logiciels en se reposant sur l'expérience de spécialiste du génie logiciel que possède

[1] Cette étude fait référence à la version 3 du langage *Eiffel*. La dernière version en date à l'heure de cette rédaction est la 4 — sortie après l'étude bibliographique présentée ici — qui apporte quelques modifications et nouveautés.

[2] C'est-à-dire qu'il ne repose pas, comme *C++*, sur un langage non objet (en l'occurrence *C*) qui permet de passer outre les *capacités objets*.

son concepteur. Les principes qui ont présidé à la genèse d'*Eiffel* sont donc : fiabilité, extensibilité, réutilisabilité et portabilité. Vous trouverez une très intéressante étude à ce propos dans [79].

Cette thèse traite des classes et des relations entre elles. Nous nous sommes donc particulièrement attachés à comprendre ces concepts au sein du langage *Eiffel* (et du langage *Java*, comme vous le verrez dans la section 2.2.2 page 30). Nous allons donc maintenant vous présenter une synthèse de nos observations à ce sujet. Nous poursuivrons ensuite en dégageant ce qui pour nous constituent les points forts et les faiblesses de ce langage.

Avant de commencer l'étude des classes *Eiffel*, nous aimerions définir le terme de *description*. Il s'agit tout simplement pour nous d'une généralisation de la notion de classe. Une description est donc une représentation, une définition d'un ou plusieurs types de données. Ainsi une classe *Eiffel* est une description mais également une interface *Java* (cf. section 2.2.2 page 30) ou encore un type primitif (tel `int`) en *C++*.

Classes et autres descriptions

En première approche, nous pouvons dire que les descriptions *Eiffel* se distingue en deux catégories :

1. **les classes :** La classe *Eiffel* est une représentation, un module et un container d'objets que l'on nomme ses instances. Elle constitue un composant de base de la structure d'une application.

   En tant que représentation, son rôle est de définir un ou plusieurs types et donc le système de typage d'*Eiffel*.

   Vu comme un module, elle décrit une unité de décomposition du logiciel. Ce module renferme des *primitives* qui sont soit des *attributs* décrivant une valeur, un état, soit des *méthodes* ou *routines* qui définissent un algorithme, un comportement.

   Enfin, en tant que container, elle agit comme un ensemble, nommé extension[3], regroupant ses instances. Cette extension contient les instances dites *directes*, qui sont d'un type décrit par la classe elle-même, mais aussi les instances *indirectes*, qui sont d'un type décrit par une sous-classe de la classe elle-même (cf. section 2.2.1 page 26 pour la notion de *sous-classe*).

   De plus, une classe *Eiffel* peut être :

   différée (ou *abstraite*) Dans ce cas, elle doit posséder au moins une primitive elle-même différée (une méthode sans corps) et ne peut créer d'instance directe (son extension ne contient donc que des instances indirectes). Une classe abstraite est donc :
   - soit une description incomplète qui modélise des objets dont, à ce niveau de conception, on ne connaît pas toutes les primitives ou tous les corps de méthode. Cette description aura vocation à être complétée en étant héritée par une classe non différée.

[3] Rappelons que le terme d'extension signifie généralement *ensemble des instances d'une description*.

– soit une description d'une entité essentiellement abstraite, dont les comportements ne peuvent donc pas être vraiment définis. Cette classe pourra alors être héritée par une classe non différée pour être concrétisée.

expansée Comme il est fréquent dans les langages à objets, les objets issus des classes *Eiffel* sont en fait conservés au travers de références attachées à des variables. Or, parfois, il est utile ou pratique d'avoir directement accès à une valeur et non pas à une référence. Cela est par exemple utile dans les cas suivants :
– Lorsque nous souhaitons créer des objets composites (au sens fort du terme, c'est-à-dire avec une dépendance de la *durée de vie* des composants par rapport à celle du composé), un accès direct aux composants plutôt qu'à une référence est préférable.
– Éviter le passage par une référence peut induire un léger gain de performance qui, s'il est multiplié une grand nombre de fois, peut s'avérer sensible.
– La mise en œuvre d'opérations de bas niveau (sur des chaînes de bits par exemple) ou de structures partagées avec d'autres langages peut profiter de l'absence de référence.
– Enfin, et peut-être surtout, la gestion des types fondamentaux s'accommode naturellement d'un accès direct à une valeur.

Ainsi, les classes expansées d'*Eiffel* ont pour principal propriété de posséder des instances qui ne sont pas des références, mais bien directement des valeurs. Une classe abstraite n'a pas d'instance, donc, par essence, une classe abstraite ne peut pas être expansée.

générique Il est également possible en *Eiffel* de déclarer une classe générique. Cela signifie que cette classe possède un ou plusieurs paramètres génériques formels. En leur donnant une valeur, nous obtenons donc un moyen de paramétrer les types représentés par cette classe. Si nous définissons, par exemple, la classe Liste avec le paramètre générique formel P, nous pourrons associer Personne à P pour créer une liste de personnes ou encore associer Chose à P pour établir une liste de choses. Cela montre la particularité essentielle d'une classe générique : contrairement à une classe non générique, une classe générique décrit plusieurs types, en fait autant qu'il existe de combinaisons de paramètres génériques effectifs différents pour elle au sein de l'application.

En *Eiffel*, une caractéristique particulièrement intéressante est la capacité de contraindre les paramètres génériques formels. Nous pouvons ainsi dire que notre liste possède une paramètre générique formel P qui est contraint au type Personne. Ainsi, nous pourrons définir des listes de personnes aussi bien que des listes d'employés ou d'enfants (qui sont tous des personnes) mais plus des listes de choses.

Une classe générique peut être abstraite ou expansée, mais pas les deux à la fois.

2. **les descriptions de base :** Outre les classes et leurs variations, nous trouvons dans *Eiffel* trois autres sortes de description.

   (a) Les **types fondamentaux** sont un élément essentiel du langage *Eiffel*. Il s'agit pour eux de décrire un type pour les entités de base que sont par

exemple les entiers, les booléens ou les caractères. Leur spécificité est qu'ils représentent simplement des valeurs et que le langage les reconnaît comme telles. Chaque type fondamental ressemble donc à une classe expansée dont les primitives seraient remplacées par des choix syntaxiques du langage lui-même (256 est syntaxiquement considéré comme un objet entier). Bien qu'aucun héritage ne soit mis en place entre types fondamentaux, un choix arbitraire mais pratique permet de considérer les entiers comme des flottants.

Pour pouvoir manipuler des instances de type fondamental comme des objets référencés, et pas seulement comme des valeurs, chaque type fondamental hérite, sans rien modifier, d'une classe non expansée. Ainsi, les instances d'un type fondamental sont également instance de sa classe ancêtre qui les présente comme des objets standards.

(b) Les **types binaires** constituent la seconde sorte. Il s'agit de pseudo-classes, potentiellement en nombre infini (autant que d'entiers naturels non nuls, en fait), qui permettent de gérer des chaînes de n bits. Par exemple, la classe BIT_8 assure la manipulation, au niveau le plus fin, d'un octet. Les types binaires constituent une particularité en *Eiffel* car il ne s'agit pas de classes comme les autres. En fait, il ne s'agit pas de classes du tout mais, dès qu'un objet de type BIT_n est déclaré, tout se passe comme si la classe BIT_n existait bel et bien. Ainsi, en déclarant deux attributs, respectivement de type BIT_8 et BIT_9, nous utilisons automatiquement les deux pseudos-classes correspondantes.

Les types binaires ne sont pas intégrés à l'arbre d'héritage. Il est cependant possible d'agir, pour des raisons pratiques, comme si chacun était un descendant de la classe racine ANY et comme si BIT_n héritait de BIT_m à chaque fois que m est strictement supérieur à n.

(c) Troisième et dernier cas particulier de description au sein du langage *Eiffel*, le **néant**, autrement dit la classe NONE est-elle aussi une pseudo-classe. Elle possède les caractéristiques suivantes :
- NONE hérite automatiquement de toutes les classes du graphe d'héritage, et cela sans tenir compte d'aucun des conflits.
- NONE n'exporte aucune primitive, c'est pour cette raison que les conflits n'ont aucune incidence.
- NONE possède une instance et une seule, appelée void. De par la place de sa classe dans le graphe d'héritage, void est compatible avec tous les types de l'application et représente la valeur associée à toute variable non initialisée.

Précisons enfin que les descriptions ARRAY et STRING possèdent quelques capacités particulières mais que ces dernières ne sont en fait sensibles qu'au niveau syntaxique. En dehors de cela, ces deux classes sont tout à fait standards.

Relations entre descriptions

Après avoir étudié les mécanismes de description présents dans *Eiffel*, nous nous attachons maintenant à décrire, pour ce même langage, quelles sont les relations entre descriptions que le programmeur peut définir.

Il en est principalement de trois sortes[7] :

La clientèle Il s'agit de la relation qui lie une classe à celle du type de ses attributs[4]. Si la classe A possède un attribut de type B, nous pouvons donc dire que A est cliente de B, au sens où elle utilise les services de cette dernière. Une classe peut être cliente de plusieurs classes, voire d'elle-même, et aussi fournisseur de plusieurs classes.

Un des mécanismes intéressants disponible pour la clientèle en *Eiffel* est l'existence des clauses d'exportation feature. Grâce à cela, le programmeur peut déclarer au sein d'une classe quelles primitives sont exportées vers (c'est-à-dire *visibles par*) quelles classes. Bien que restrictif et limité — une classe ne peut pas connaître *a priori* l'ensemble de ses clientes potentielles — ce dispositif est pratique pour spécifier un peu plus précisément l'interface d'une classe. Ce mécanisme nuit malheureusement à la lisibilité du code d'une classe car il est parfois difficile de déterminer quelles primitives sont effectivement visibles.

Il est possible pour une classe d'être cliente d'une classe expansé ou d'une classe générique, voire d'une classe expansée générique. Il est aussi possible de déclarer une clientèle expansée vers une classe qui ne l'est pas. Enfin, déclarer une clientèle expansée vers une classe expansée n'est en aucun cas une gêne.

L'héritage Seconde relation traditionnelle des langages à objets, après la clientèle, l'héritage du langage *Eiffel* recèle plusieurs particularités. Le but de l'héritage est double :

1. En premier lieu, il s'agit d'un système d'*extension de module*. Hériter d'une classe permet d'en créer une autre (souvent nommée sous-classe) et de définir un nouveau comportement en fonction de celui de la classe héritée. Il est envisageable d'ajouter ou de modifier des primitives. La suppression est par contre impossible en regard du point suivant.
2. D'un second point de vue, l'héritage permet de *spécialiser* une classe, c'est-à-dire que la classe héritière est une version affinée de la classe héritée. Dans l'esprit qu'un objet spécialisé doit savoir au moins répondre à toutes les requêtes d'un objet plus général, la suppression d'une primitive s'en trouve donc non applicable par héritage. Cette idée d'affinage est le socle du système de type. En *Eiffel*, il est en effet considéré qu'une classe héritière décrit un sous-type de celui défini par la classe héritée. Il s'agit d'un principe de base relevant du polymorphisme d'inclusion.

L'héritage d'*Eiffel* est multiple, c'est-à-dire qu'une classe peut hériter de plusieurs autres[5]. Si elle hérite plusieurs fois de la même classe, l'héritage est alors dit répété. Toute classe hérite par défaut de la classe racine ANY si aucun héritage n'est signalé. De plus, la classe prédéfinie NONE hérite automatiquement de toutes les classes de l'application. Rappelons que NONE n'exporte aucune primitive et peut donc se permettre d'ignorer tous les conflits éventuels.

Le but principal de l'héritage est de bénéficier automatiquement de la définition des primitives des classes héritées. Cependant, l'importation telle quelle de ces primitives n'est pas toujours souhaitée. Dans ces clauses d'adaptation, *Eiffel* permet donc d'intervenir sur les primitives héritées :

[4]Par extension, la notion de clientèle peut être élargie aux paramètres de méthode et aux résultats de fonction.

[5]Une classe ne peut pas hériter d'elle-même.

- Il est ainsi possible de *renommer* une primitive héritée pour lui associer un nouveau nom plus conforme au comportement présent dans la classe héritière ou pour éviter un conflit.
- Le programmeur peut également *redéfinir* une primitive héritée. La redéfinition consiste à donner un nouveau corps, une nouvelle implémentation, à la primitive. Ainsi, il est possible de spécialiser un algorithme ou encore de spécialiser le type d'un attribut ou des paramètres d'une méthode. L'héritage d'*Eiffel* est donc dit covariant puisque les signatures des primitives se spécialisent au fur et à mesure de la spécialisation des classes qui les contiennent.
- Une troisième adaptation possible consiste à concrétiser une primitive. Lorsque celle-ci est abstraite dans la classe héritée, il est donc permis de lui donner ainsi une implémentation dans la classe héritière. Si une ou plusieurs concrétisations ont pour effet d'éliminer toutes les primitives abstraites d'une classe différée, alors celle-ci devient concrète. Pour des raisons pratiques, il est aussi possible de rendre abstraite lors d'un héritage une primitive qui était concrète. Dans ces deux cas, la covariance s'applique aussi.
- La *fusion* permet d'unifier — toujours en respectant la covariance — deux primitives venant de deux classes héritées. Il est nécessaire pour cela que les signatures soient compatibles et que les deux primitives héritées soient abstraites.
- Enfin, la sélection permet de résoudre le problème posé par l'héritage dit en diamant. Imaginons une classe A définissant une primitive P, une classe B hérite de A et redéfinit et renomme P en PB et une classe C hérite aussi de A et redéfinit et renomme aussi P en PC. Et maintenant créons une classe D qui hérite de B et de C. Si on applique à l'instance d de D la primitive PB ou PC, tout va bien. Mais si, en profitant du polymorphisme, on affecte d à une variable de type A et que l'on souhaite lui appliquer P, nous ne saurons quelle version de P choisir : PB ou PC ? La sélection permet tout simplement de choisir la primitive à appliquer si un pareil cas se présente.

L'héritage a également des conséquence sur les assertions, que ce soit les préconditions ou les postconditions de méthode ou encore les invariants de classe. L'idée est toujours la même : la classe héritière doit offrir au moins tous les services de la classe héritée. Nous pouvons en déduire que les préconditions de l'héritière doivent être équivalentes ou plus faibles[6] que celles de l'héritée. Alors que les postconditions et invariants de l'héritière doivent être équivalents ou plus forts que ceux de l'héritée. Cela est cohérent avec le principe de covariance évoqué précédemment.

Nous avons vu les clauses d'exportation qui permettent d'affiner la volonté du programmeur du point de vue de la relation de clientèle. Toute classe qui hérite peut modifier le choix d'exportation pour chaque primitive héritée, sachant que, par défaut, celui-ci est constant. Nous voyons ainsi une influence en *Eiffel* de l'héritage sur la clientèle.

La généricité Troisième et dernière relation entre classes que nous trouvons en *Eiffel*, la généricité est peut-être la plus représentative de ce langage.

La généricité lie une classe générique, par exemple TABLEAU(E), avec les classes qui représentent les types des paramètres génériques effectifs, qui viendront prendre la place du E. Ainsi, si au cours de l'application est créé un tableau de

[6] Nous disons qu'une condition est plus faible qu'une autre si la seconde implique la première.

personnes (TABLEAU(PERSONNE)), une relation de généricité lie alors TABLEAU à PERSONNE.

La généricité d'*Eiffel* se rapproche beaucoup plus de la clientèle que de l'héritage au sens où il s'agit d'utiliser les services d'une classe et non de les importer. L'une de ses caractéristiques les plus intéressantes est de pouvoir être contrainte. Ainsi, si nous déclarons la classe générique TABLEAU(E->ELEMENT_TABLEAU), les seules paramètres génériques formels acceptables pour TABLEAU sont ELEMENT_TABLEAU et ses descendants.

Autres éléments

Nous décrivons dans cette section quelques fonctionnalités d'*Eiffel* intéressantes en plus de ses descriptions et relations entre elles.

- *Eiffel* a été conçu pour être utilisé depuis la phase d'analyse jusqu'à celle de maintenance en passant par la conception et la programmation. La structuration des classes et le mécanisme d'assertions devait permettre ce très intéressant objectif. Cependant nous pensons que, dans les faits, *Eiffel* reste principalement un langage de programmation.
- Le système d'assertions demeurent une des grandes qualité d'*Eiffel* : il améliore en effet sensiblement la définition, la fiabilité et la documentation des classes et donc des applications.
- Un mécanisme d'exceptions réifiées offre un support notable pour la gestion des cas anormaux.
- Les types ancrés, c'est-à-dire déclarés comme étant similaires à un autre type, permettent d'élégants apports et limitent souvent les redéfinitions inutiles.
- *Eiffel* dispose de méthode à usage unique (mot-clé once). La première fois qu'elles sont appelées, leur corps est effectivement exécuté, les fois suivantes non. S'il s'agit de fonction le résultat calculé la première fois est rendu les fois suivantes sans nouveau calcul.
- Les règles de *conformité de type* d'*Eiffel* sont précises mais complexes. Nous disons que le type TA décrit par la classe A est conforme au type TB décrit par la classe B si et seulement si :
  - TA et TB sont identiques ou
  - A hérite de B et TA n'est ni expansé, ni un type binaire et B est non générique ou
  - A est le néant et TB n'est ni expansé, ni un type binaire ou
  - A et B sont identiques et génériques et chaque paramètre générique effectif de TA est conforme à son correspondant dans TB ou
  - A hérite de B qui est générique et, pour chaque paramètre générique formel de A qui est aussi un paramètre générique formel de B, alors le paramètre générique effectif de TA doit être conforme au paramètre générique effectif de TB[7] ou
  - TA est conforme à TC (type quelconque) qui est conforme à TB et TB n'est ni expansé, ni un type binaire ou
  - TB est identique à TA expansé ou

[7] Cette phrase résume trois pages du livre de Bertrand Meyer [78], sa simplicité est donc toute relative...

– A est INTEGER et B est REAL ou
– A est INTEGER ou REAL et B est DOUBLE ou
– TA est un type binaire et B est ANY ou
– TA est un type binaire BIT_m et TB est un type binaire BIT_n et $m \leq n$.

Cette analyse démontre la complexité qui peut se cacher sous un mécanisme *a priori* simple comme l'héritage. Modifier, adapter un tel mécanisme est donc un travail complexe et fastidieux. Nous nous attacherons dans notre étude à faire de cette adaptation une chose raisonnable.

2.2.2 *Java*

C++ [91, 92] a vu le jour en 1983. Il a été le langage à objets le plus utilisé par les programmeurs jusqu'à l'avènement de *Java*[8] [32, 21, 55, 3, 61, 65, 64]. Ce dernier a eu notre préférence lors de notre étude bibliographique principalement en raison de sa structure purement *à objets* (si on met de côté les types de base) là où *C++* conserve une compatibilité avec le langage *C*[9].

De *C++*, *Java* a globalement hérité deux choses : sa syntaxe et sa popularité. Syntaxiquement *Java* peut en effet être considéré comme une extension de *C*. Du point de vue de son utilisation, *Java* semble peu à peu remplacer son illustre prédécesseur. *Smalltalk* aussi fait sentir son influence sur la création du langage *Java*. On y retrouve notamment les idées de ramasse-miettes (aussi présent dans *Eiffel*) et de machine virtuelle.

Dans les trois sections qui suivent, nous allons donc étudier les caractéristiques de *Java* relatives à nos travaux.

Classes et autres descriptions

Comme nous l'avons fait avec *Eiffel*, nous discutons ici des sortes de classes et d'autres descriptions de *Java*. Nous ferons d'ailleurs fréquemment une comparaison avec l'étude qui vient d'être faite d'*Eiffel*, de manière à garder à l'esprit les deux approches et à ne pas répéter longuement les points communs.

Nous trouvons en *Java* deux grandes sortes de descriptions (les classes et les interfaces) et deux plus structurelles (les tableau et les types primitifs) que nous traiterons ensemble.

1. **les classes :** Commençons par les classes. Comme pour *Eiffel*, une classe *Java* peut être considérée comme :
   – une **représentation** d'un type,
   – un **module**, élément de construction d'une l'application et
   – un **container** de ses instances.

   De plus, une classe *Java* peut être :

   abstraite Elle représente alors un concept abstrait ou à l'implémentation incomplète. Comme son équivalent *Eiffel*, elle sert souvent de racine à un

[8] Cette étude fait référence à la version 1.3 du langage *Java*, dernière version en date à l'heure de cette rédaction. Une version 1.4 est annoncée pour les mois à venir.

[9] Nous ne discutons pas ici des avantages et inconvénients d'avoir choisi de rester compatible avec *C*, nous indiquons simplement que cette caractéristique nous a fait préférer *Java*.

sous-arbre de classes qui concrétisent la partie abstraite de différentes façons.

interne Cette possibilité, apparue dans les versions récentes du langage, permet de représenter des sous-classes[10], c'est-à-dire des classes qui sont décrites *à l'intérieur* de la définition d'une autre classe. Cette particularité n'existe pas en *Eiffel*. Les genres de classe interne présents en *Java* sont nombreux. Nous trouvons :
- les classes membres, dont chaque instance est associée à une instance de la classe englobante, et qui peuvent être concrètes ou abstraites,
- les classes membres statiques (sortes de *classe de classe* si nous reprenons la terminologie d'*attribut de classe*), concrètes ou abstraites,
- les classes locales, concrètes ou abstraites, qui sont locales à une méthode ou encore
- les classes anonymes qui n'ont pas besoin de nom puisqu'elles sont locales à une expression et n'ont donc de sens et d'effet qu'à l'emplacement de leur définition.

À la différence d'*Eiffel*, les classes génériques n'existent pas en *Java*, ce qui est bien dommage. L'une des conséquences notables de cela est qu'une classe *Java* représente un type et un seul là où une classe générique *Eiffel* peut en représenter plusieurs.

Enfin, précisons que les classes *Java* admettent la surcharge, c'est-à-dire que deux méthodes de même nom peuvent apparaître dans une classe pour peu que le nombre et/ou le type de leurs paramètres diffèrent.

2. **les interfaces :** Poussant jusqu'au bout le concept amorcé par les classes abstraites, les créateurs de *Java* ont équipé ce dernier de la notion d'interface.

   Une interface diffère d'une classe parce que :
   - Elle ne peut pas posséder d'instance directe, comme une classe abstraite.
   - Elle ne peut pas définir de constructeur.
   - Toutes ses primitives sont publiques.
   - Toutes ses méthodes sont abstraites.
   - Seuls les attributs de classe constants sont acceptés.
   - Des relations entre descriptions spécifiques aux interfaces existent, telles l'héritage inter-interface et l'implémentation (cf. section 2.2.2 page suivante).

   Ainsi une interface décrit un type de données abstrait qui pourra, par la suite, être implémenté par une ou plusieurs classes. Ces classes devront respecter, et éventuellement étendre, la définition donnée par l'interface qui agit donc telle une API publique et abstraite. Un schéma de descriptions classiques en *Java* se compose d'interfaces implémentées et complétées par des classes abstraites elles-mêmes concrétisées et complétées par des classes concrètes.

   Il est possible de définir des interfaces internes, à l'image des classes internes. Mais seul le genre *interface membre statique* est permis. Ni les interfaces membres, ni les interfaces locales, ni les interfaces anonymes n'existent car elles n'auraient pas grand sens.

   Les interfaces admettent, comme les classes, la surcharge.

3. **les descriptions de base :** Nous trouvons en *Java* deux sortes de descriptions de base : les tableaux et les types primitifs.

[10] En anglais, le terme d'*inner class* est généralement employé.

(a) Les **tableaux** forment la structure de données bien connue permettant de gérer des objets contenus dans cette structure au travers d'un indice spécifiant le numéro d'ordre de l'élément courant. Les tableaux forment un ensemble de descriptions virtuelles assimilables à des classes qui apparaissent au gré de la création de leurs instances. Par exemple, la déclaration d'un attribut de type Chose() permet de lier cet attribut à un objet qui sera décrit comme appartenant à la pseudo-classe Chose() (tableau de Choses). Cette manière particulière de gérer les tableaux pallie en partie l'absence de généricité pour les classes. En effet, les tableaux sont, de fait, des containers génériques. Cependant, seuls les tableaux bénéficient de cette apport générique et encore cela leur donne-t-il un caractère particulier qui les différencie des types représentés par les classes et interfaces. Terminons en précisant que les tableaux *Java* peuvent également être manipulés au travers d'une classe réflexive les décrivant : Array.

(b) Les **types primitifs** de *Java* sont les équivalents des types fondamentaux d'*Eiffel*. Plus nombreux que ces derniers (nous en dénombrons par exemple quatre rien que pour les entiers), ils sont en fait grandement inspirés de ceux du langage C. Citons cependant parmi eux le type boolean qui fait défaut à C. Comme en *Eiffel*, une classe correspondant à chaque type fondamental existe : elle permet de voir les valeurs fondamentales comme des objets en les encapsulant dans une structure adéquate.

Relations entre descriptions

Passons maintenant aux relations entre descriptions. Nous en trouvons de trois sortes :

1. **L'héritage** Il faudrait en fait dire les héritages car nous pouvons en discerner trois relativement différents. Avant de les détailler, précisons que comme pour *Eiffel*, l'héritage est ici un moyen d'étendre un module et de spécialiser une classe. De plus, tout classe *Java* hérite par défaut de la classe racine Object si aucun héritage n'est signalé.

   Contrairement à *Eiffel*, *Java* ne dispose pas d'un mécanisme de clause d'adaptation. Il est donc impossible de renommer une primitive mais la limitation qui a pour cause l'absence de clause s'arrête là. En effet, la redéfinition et la concrétisation peuvent être effectuées sans nécessité de les déclarer. La fusion et la sélection ne présentent aucun intérêt dans ce contexte car l'héritage multiple ne s'applique, en *Java*, qu'entre primitives abstraites.

   Voyons donc ces héritages :

   héritage entre classes Ce cas est valide lorsqu'une classe concrète ou abstraite hérite d'une classe concrète. La classe héritière est considérée comme une sous-classe, une descendante de la classe héritée. Toutes les primitives de la classe héritée sont automatiquement présentes dans chaque classe héritière, à l'exception des primitives privées (cf. section 2.2.2 page 34), des constructeurs et des initialiseurs.

   Il est possible de déclarer une classe finale (mot-clé final), cela empêche de déclarer une classe qui en hérite.

L'héritage *Java* considère que le programmeur ne peut pas redéfinir les attributs (c'est possible en *Eiffel*). Donc, si nous déclarons un attribut att dans une classe A qui hérite d'une classe B dans laquelle un att est déjà présent, les objets de type A verront leur att avec son type et les objets de type B verront leur att avec son type et il n'y a aucune contrainte à respecter entre les types des att. Nous parlons alors de remplacement (*hiding*). Pour les méthodes, les redéfinitions sont évidemment (*overriding*) autorisées mais il faut prendre garde à réellement effectuer une redéfinition et à ne pas tomber tout simplement dans un cas de surcharge (*overloading*). Ce fait implique la non-variance non seulement du nombre mais aussi du type des paramètres de méthode lors de la redéfinition là où *Eiffel* utilisait la covariance. Par contre, les types de résultat de fonction doivent être non-variants. De plus, ils n'entrent pas dans la discrimination permettant la surcharge.

héritage entre interfaces L'héritage qui existe entre les classes peut également être placé entre interfaces. Dans ce cas, il possède les caractéristiques suivantes :

- Il est multiple, c'est-à-dire qu'une interface peut hériter de plusieurs autres. Cela ne pose en effet aucun problème sérieux de conflit : les primitives définies une seule fois sont simplement importées, celles qui sont décrites de la même manière dans plusieurs interfaces héritées sont fusionnées. En plus d'être multiple, cet héritage est potentiellement répété, bien que cela soit sans grand intérêt.
- Une interface ne peut pas être déclarées final pour empêcher son héritage.
- Comme pour l'héritage entre classes, le renommage n'est pas possible. La fusion et la sélection ne servirait à rien compte tenu de l'absence systématique de corps pour les méthodes. La concrétisation est non applicable de manière évidente. Seule la redéfinition est possible bien que d'un intérêt fort limité.

concrétisation Dernier de la série d'héritages de *Java*, nous souhaitons présenter ce cas particulier à part bien qu'il puisse être facilement intégré à l'héritage entre classes. Nous nous trouvons ici dans la situation d'une classe concrète qui hérite d'une classe abstraite. La principale différence que nous souhaitons mettre en exergue est l'obligation de donner un corps, dans la classe héritière, aux méthodes qui sont abstraites dans la classe héritée. Il aurait été également possible de différencier ce cas de l'héritage pour le langage *Eiffel*. Cette liberté dans le choix de la granularité de ce qui est décrit se retrouvera tout au long de notre définition du travail d'un méta-programmeur.

2. **L'implémentation** Il s'agit de la relation qui permet de lier une ou plusieurs interfaces à une classe : cette dernière implémente les premières, c'est-à-dire les concrétise complètement (si la classe est concrète) ou en partie (si elle est abstraite).

   Chaque classe peut implémenter plusieurs interfaces et doit donner à chaque méthode, soit le statut de méthode abstraite (et alors la classe le sera forcément aussi) soit un corps (et, si toutes les méthodes ont ainsi une implémentation, la classe peut être concrète). La classe ne peut implémenter qu'une seule fois chaque interface, du moins directement. Ainsi si l'interface I1 est héritée par l'in-

terface I2 qui est elle-même implémentée par la classe C, cette dernière ne peut pas implémenter une deuxième fois I2, mais peut implémenter une deuxième fois (la première est indirecte) I1.

3. **L'agrégation et la composition** L'agrégation de *Java* est peu ou prou l'équivalent de la clientèle d'*Eiffel*, la composition de *Java* étant à rapprocher plutôt à la clientèle expansée d'*Eiffel*.

   L'agrégation lie donc notamment une classe avec les classes de ses attributs, à la condition que ces attributs référencent des objets. Si un attribut est d'un type primitif, alors nous parlons de composition. En effet, dans ce dernier cas, la durée de vie de l'entité attachée à l'attribut est dépendante de celle de l'objet possédant cet attribut[11]. Donc pour résumer, un attribut vers un objet décrit une agrégation alors qu'un attribut vers une entité primitive désigne une composition[12]

   Signalons enfin qu'un attribut, qu'il s'agisse d'agrégation ou de composition, peut être *d'instance* ou *de classe*.

Autres éléments

Comme nous l'avons fait pour *Eiffel*, nous terminons cette section sur *Java* en citant d'autres caractéristiques de ce langage qui nous ont intéressés.

- Les classes et les interfaces *Java* peuvent être qualifiées par plusieurs mots-clés appelés *qualifieurs* :

  abstract Il signale une classe abstraite. Ce qualifieur est automatique et facultatif pour une interface.

  final La classe ne peut pas être héritée. final n'est évidemment pas permis pour une interface.

  public La classe ou l'interface est visible partout où son paquetage l'est.

  static Réservé aux classes internes, il permet de déclarer une classe membre statique. Une interface membre statique possède automatiquement le qualifieur static et il est interdit de le spécifier.

  strictfp Ce qualifieur peu utilisé indique que toutes les méthodes de la classe doivent effectuer tous leurs calculs en virgule flottante conformément au format IEEE 754. strictfp n'est évidemment pas permis pour une interface.

  rien La classe ou l'interface est visible seulement au sein de son paquetage.

- Les primitives *Java* aussi peuvent être qualifiées :

  abstract Réservé aux méthodes, il signale qu'aucun corps n'est spécifié. La description englobante est alors forcément également abstraite.

  final Appliqué à une méthode, il empêche sa redéfinition. Sur un attribut ou une variable (variable locale ou paramètre), il interdit la modification de sa valeur après initialisation.

[11] Il s'agit d'une vue de l'esprit : la durée de vie de 3 n'est pas dépendante d'un objet Personne qui signale ainsi le nombre de ses enfants. Cependant, conceptuellement, le nombre d'enfants de Personne n'existera plus si ce dernier objet est supprimé.

[12] Ce choix terminologique entre *agrégation* et *composition* est généralement admis dans la communauté scientifique mais reste sujet à discussion car arbitraire.

native Il indique, exclusivement pour une méthode, que son implémentation est réalisée spécifiquement pour une plate-forme, souvent en *C*.

private La primitive est visible seulement au sein de sa classe.

protected La primitive est visible seulement au sein de son paquetage et au sein de toutes ses sous-classes, indépendamment de leur paquetage.

public La primitive est visible partout où sa classe l'est.

static Pour un attribut ou une méthode, il indique qu'il(elle) fait référence à une classe et non à ses instances ou, en d'autres termes, qu'il(elle) est partagé(e) par toutes les instances de cette classe. Appliqué à un initialiseur, il signale que son exécution se fait au chargement de la classe plutôt qu'à la création d'une instance.

strictfp Utilisé seulement sur une méthode, il indique que tous ses calculs en virgule flottante doivent être conforme au format IEEE 754.

synchronized Ce qualifieur impose que la méthode visée soit considérée comme atomique par le système multi-thread. Pour une méthode de classe, c'est la classe qui est verrouillée, sinon c'est l'instance considérée.

transient L'attribut concerné ne doit pas être sérialisé si l'objet qui le réfère l'est. En d'autres termes, nous pouvons considérer qu'un objet persistant signale ainsi lesquels parmi ses attributs ne doivent pas être persistants.

volatile Ce qualifieur, d'un usage particulièrement anecdotique, signale que l'attribut ne doit pas faire l'objet d'optimisation car il pourrait être utilisé par plusieurs threads non synchronisés. Il se rapproche ainsi de synchronized.

rien La primitive est visible seulement au sein de son paquetage.

– *Java* dispose de trois sortes de primitives particulières :

Les constructeurs Ils portent le nom de la classe qui les définit et servent à initialiser un objet juste après son allocation. Ils ne sont pas hérités et font l'objet d'une politique de *chaînage* spécifique, chacun appelant celui de la classe héritée selon certaines règles. Faute de définition explicite d'un constructeur, il en existe un par défaut, dans chaque classe, qui ne définit aucun paramètre et ne fait qu'appeler celui de la classe héritée.

Les initialiseurs Ils ne portent pas de nom et ce sont d'ailleurs les seules primitives anonymes. Ils servent à mettre en place des initialisations automatiques d'attribut entre l'allocation de l'objet et l'appel au constructeur. Ils sont par exemple utiles pour factoriser une initialisation qui devrait être systématiquement réalisée par tout constructeur. Lorsqu'il est déclaré static — c'est donc un initialiseur de classe — l'initialiseur peut offrir l'opportunité de donner une valeur aux attributs de classe, par exemple. L'usage des initialiseurs reste peu fréquent.

Les finaliseurs Pendant des constructeurs, ils servent à réaliser des opérations spécifiques (par exemple, la fermeture d'un fichier) avant la désallocation d'un objet[13]. Au contraire des constructeurs, il ne peut y avoir qu'un seul finaliseur par classe (il se nomme finalize) et il ne doit pas posséder de paramètre. Le finaliseur est appelé automatiquement avant l'action du

[13]Cette désallocation est automatique avec un système de ramasse-miettes.

ramasse-miettes mais il n'existe aucune politique de chaînage : tout doit être géré par le programmeur. De fait, les finaliseurs, très utiles dans certains contextes, sont très peu utilisés. Il convient de le faire de toute façon avec prudence car ils génèrent parfois des comportements pour le moins audacieux, comme la résurrection d'un objet perdu mais pas encore libéré.

2.3 Protocoles méta-objets

Nous venons de donner un aperçu des langages de programmation orientés-objets *Eiffel* et *Java* qui sont principalement à l'origine de nos réflexions concernant les relations entre classes. Nous allons maintenant citer différents systèmes et langages, divers modèles et techniques qui offrent au programmeur ou au méta-programmeur la capacité d'agir sur ces relations. L'objet de cette section porte ainsi sur les protocoles méta-objets. Nous avons étudié les protocoles méta-objet car c'est la voie qui a été choisie pour apporter des solutions aux problèmes que nous traitons. Pour cela, nous nous sommes intéressés à plusieurs systèmes tels *CLOS-MOP*, *Smalltalk*, *OpenC++*, *OpenJava*, *Javassist*, *metaXa*, *Iguana* et *FLO*.

Tout d'abord, définissons le concept de protocole méta-objet. Il s'agit, dans la vision très générale qui est la nôtre, d'un système, d'un ensemble d'opérations ou de composants logiciels permettant de représenter et d'adapter le comportement d'un langage orienté-objets.

2.3.1 *CLOS-MOP*

CLOS [68, 70, 63] est une sur-couche orientée objets de *Common Lisp*. Il est compatible avec ce dernier et offre en plus les notions de classe, d'objet, d'héritage, ... Mais là n'est pas pour nous le plus important. *CLOS* est en effet accompagné d'un protocole méta-objets nommé *MOP* qui permet d'intervenir au niveau de la méta-programmation et donc de modifier le comportement de *CLOS*. Voici les points les plus importants de *CLOS* et *MOP* dans le contexte de cette étude :

- *CLOS-MOP* réifie principalement cinq concepts :
  1. Les classes sont instances de méta-classes qui définissent leur structure et leur comportement.
  2. Les attributs sont représentés au moyen d'un méta-objet qui décrit leur définition.
  3. Les méthodes sont également l'objet d'une réification.
  4. La composition de méthode (qui permet notamment d'appliquer les before, after et around) est réifiée.
  5. Enfin, l'une des entités les plus importantes est formée par les fonctions génériques qui jouent un rôle d'interface pour les méthodes. Une fonction générique décrit une opération générale (par exemple *calculer une somme*) qui sera implémentée grâce à plusieurs méthodes, en fonction de la classe des paramètres de cette opération.
- *CLOS* permet l'utilisation d'un héritage multiple, c'est-à-dire que chaque classe peut hériter de plusieurs autres. La gestion des conflits amenés par cet héritage est réalisée de manière simple : une liste de précédence est calculée pour chaque classe. Cette liste cite les classes héritées dans l'ordre qui devra être suivi

pour la recherche de primitive : la première primitive trouvée qui possède la bonne signature sera déclenchée. L'ordre de cette liste de précédence peut être modifié par le programmeur pour réaliser une politique de liaison dynamique particulière.
- Chaque méthode peut être complétée par trois sortes d'ajout. Une pré-méthode (before) sera forcément exécutée avant sa méthode. Les pré-méthodes peuvent être cumulées et sont alors lancées dans l'ordre inverse de leur déclaration. Une post-méthode (after) représente évidemment ce qui est exécutée après la méthode principale ; les post-méthodes se lancent dans l'ordre de leur définition. Il existe enfin des méthodes dites around qui permettent de compléter un comportement en réalisant l'appel de la méthode au sein même d'un algorithme et non avant ou après. Ces politiques d'exécution sont redéfinissables.
- *MOP* associe à chaque classe *CLOS* une méta-classe qui définit son comportement. Ainsi, il est très facile d'associer à un ensemble de classes (qui auront la même méta-classe) une caractéristique spécifique. Le (méta-)programmeur dispose donc, par exemple, de la possibilité de créer une méta-classe dont les instances — des classes — comptent leurs instances. standard-class est la méta-classe par défaut de toute classe (y compris elle-même), standard-object étant la racine de l'arbre d'héritage.
- Les méthodes (de type method) ne sont pas encapsulées dans les classes. Un système de fonctions génériques (de type generic-function) est mis en place. Une fonction générique regroupe toutes les méthodes d'un même nom. Lors de la recherche de la méthode à appliquer — liaison dynamique — c'est la fonction générique qui est en fait appelée. Un algorithme standard est lancé qui, en fonction du type des paramètres obligatoires[14] et de la liste de précédence impliquée par l'héritage, choisit la méthode à appliquer. Il est possible de redéfinir en tout ou en partie l'algorithme de recherche de la primitive adéquate.
- Bien que *MOP* soit ouvert, il peut s'avérer difficile, voire très difficile, de définir de nouvelles relations inter-classes ou de modifier de manière importante les relations existantes. En effet, les points d'entrée de *MOP* dans *CLOS* ont été prévus pour adapter l'héritage et l'agrégation et non pour les remplacer ou les compléter par d'autres sortes de relations. Un méta-programmeur de talent pourra évidemment toujours y parvenir mais il devra alors réécrire tous ou presque tous les algorithmes qui forment le *MOP*.

Pour finir, nous présentons les points forts et les faiblesses de l'association *CLOS-MOP*. Nous parlons de *faiblesses* dans le cadre de notre objectif — l'adaptation des relations entre classes — pas de défauts de nature générale.

CLOS présente l'avantage d'être accessible au travers d'un seul langage, pour la méta-programmation comme pour la programmation. Le système de fonctions génériques est lui aussi intéressant car il permet à la liaison dynamique de se réaliser par l'intermédiaire de plusieurs entités (les paramètres) plutôt que d'une seule (l'objet courant) dans un langage plus traditionnel. *MOP* offre de plus de nombreux points d'entrée qui permettent de modifier le comportement de *CLOS* principalement en redéfinissant des méthodes de niveau méta.

En ce qui concerne les faiblesses, nous pouvons citer l'absence de séparation des phases de méta-programmation et de programmation, qui nous semble être une cause importante de problèmes. En effet, la capacité de modifier un langage pendant son utilisation fournit certes une liberté motivante mais reste néanmoins une énorme

[14] Les méthodes *CLOS* peuvent décrire des paramètres optionnels.

source d'erreurs et de complexité. Ajoutons également que, s'il est possible de redéfinir de nombreux comportements, et notamment ceux des relations inter-classes, il est très difficile de modifier en profondeur ces relations car cela nécessite de réécrire, ou tout au moins de compléter, des algorithmes méta qui sont loin d'être évidents. Cet inconvénient est minoré par la relative simplicité du langage support *Common Lisp* mais cette simplicité a elle-même un coût : l'absence de typage. Cette absence limite le champ des contrôles réalisables par l'interprète ou le compilateur et reporte donc à l'exécution bon nombre d'erreurs.

Le problème principal reste cependant, d'après nous, l'absence de méta-objet représentant les relations inter-classes. En effet, les classes sont réifiées par des méta-classes mais les relations ne le sont pas, ce qui implique la dispersion des algorithmes les mettant en oeuvre.

2.3.2 *Smalltalk*

Smalltalk [58, 57, 74], célèbre langage à objets, possède lui aussi une couche méta. En voici une description :

- *Smalltalk* met en place un mécanisme d'héritage simple : chaque classe hérite d'une autre, excepté la racine Object.
- Chaque classe *Smalltalk* est instance d'une méta-classe et d'une seule, cette dernière étant créée lors de la génération de sa classe. De plus, chaque méta-classe ne possède qu'une seule instance. Nous avons donc affaire ici à une bijection entre classe et méta-classe, ainsi le graphe d'héritage des méta-classes suit fidèlement et automatiquement celui des classes, ce qui présente l'avantage d'une grande simplicité. La méta-classe décrit la classe et sa structure au travers d'attributs et de méthodes. Nous y retrouvons notamment des comportements globaux tels les primitives d'entrée-sortie ou du ramasse-miettes.
- Chaque méta-classe est également le siège naturel de toutes les primitives partagées par toutes les instances (attributs et méthodes de classe). Par exemple, la méthode new qui crée une instance est un membre de la méta-classe.
- En raison de la bijection entre classe et méta-classe, une méta-classe n'a pas besoin de nom car elle reste toujours accessible par la primitive class appliquée à sa classe. Toutes les méta-classes sont instances de Metaclass.
- Une classe Behavior, qui constitue la racine méta, décrit les comportements d'une classe. Elle comporte quelques primitives permettant de naviguer dans les graphes d'héritage et d'instanciation. Par là même, elle offre un moyen d'adapter le comportement de ces deux relations mais dans une mesure limitée cependant car il n'existe pas de méta-objet pour ces relations.

Smalltalk dispose de capacités de réflexion très poussées. Les classes, les méthodes et les exceptions sont, par exemple, réifiées. Mais on trouve aussi, et cela est plus original et intéressant, une réification des processus, de la mémoire et même celle des arbres syntaxiques, voire de la pile d'exécution ou du compilateur !

Tout cela fait de *Smalltalk* l'un des langages à objets réflexifs les plus complets qui soient. De nombreux travaux se sont ainsi basés sur *Smalltalk* tels *ClassTalk* [34, 33], *NeoClassTalk* [89] et *MetaClassTalk* [6].

Comme nous venons de le voir rapidement, les capacités méta de *Smalltalk* sont relativement étendues. Elles permettent de réaliser de nombreux ajustements du comportement du langage sans pour autant remettre en cause les principaux choix sur lesquels il repose. Par exemple, modifier le comportement de l'héritage est as-

sez facilement faisable, alors qu'ajouter une nouvelle sorte de relation entre classes relèverait d'un travail d'une toute autre ampleur. En effet, rien (c'est-à-dire aucune réification des relations) n'est prévu en ce sens. Comme pour *CLOS* nous regrettons également la fusion des phases de programmation et méta-programmation ainsi que l'absence d'un typage fort, *Smalltalk* reposant sur une philosophie de contrôle exclusif à l'exécution, qui n'est pas la nôtre.

2.3.3 *OpenC++*

OpenC++ [22, 23] est un protocole méta-objet pour le langage *C++*. Contrairement à l'approche de *CLOS-MOP* et de *Smalltalk* que nous venons de voir, *OpenC++* agit à la compilation pour créer un nouveau langage. C'est-à-dire que le niveau méta ne se retrouve pas lors de l'exécution. Si cela limite la flexibilité d'*OpenC++*, cela augmente d'autant les performances des programmes qu'il décrit.

OpenC++ est un système qui compile un source *C++* étendu vers du *C++* standard. Pour cela, il utilise des méta-entités (méta-classes, méta-méthodes, méta-variables, ...) qui décrivent un composant du langage avec son comportement particulier et le reporte — déclenchement de la méthode CompileSelf de chaque méta-entité — en produisant du code *C++* standard spécifique. Ainsi, les modifications de la sémantique de *C++* sont propagées à tous les points appropriés du programme sans nécessiter une intervention forcément fastidieuse du programmeur. Le méta-programmeur doit définir, pour chaque méta-entité, une partie statique (*compile-time*) et une partie dynamique (*runtime*). La première forme une bibliothèque de méta-objets, la seconde exprime les actions de traduction requises vers *C++* standard.

Les quatre types principaux de méta-objets pris en compte par *OpenC++* sont :

Ptree Il modélise, sous forme de listes imbriquées, le code du programme, tout d'abord écrit en *OpenC++* puis devant être traduit en *C++*.

TypeInfo Il réifie la notion de type au sein d'un programme en *C++*. Il gère aussi bien les types définis par une classe que les types pointés, les types référencés et les types primitifs.

Environment Il décrit l'environnement d'un programme *C++* s'exécutant. Cette partie est nécessaire car *OpenC++* agit à la compilation et non à l'exécution. Redéfinir les primitives d'Environment permet de modifier le comportement de l'héritage et de l'agrégation. Les points d'accès sont cependant peu nombreux et nécessitent de réécrire en grande partie les algorithmes associés, ce qui est un très gros travail. De plus, l'ajout d'une nouvelle sorte de relation n'est absolument pas pris en compte.

Class Il s'agit de la méta-classe par défaut, celle qui sert également de racine au graphe d'héritage des méta-classes. Elle est chargée de compiler ses instances — des classes *OpenC++* — en classes *C++*. Le travail du méta-programmeur consiste principalement à spécialiser Class pour spécifier le comportement voulu.

OpenC++ dispose d'un mécanisme de *before* reprenant l'idée de *CLOS*. Cependant, une seule pré-méthode est admise par méthode. Un autre système permet d'associer un intermédiaire (*wrapper*) à chaque méthode pour indiquer un traitement particulier. Par exemple, nous pouvons ainsi réaliser des tâches spécifiques qui encapsulent l'exécution des primitives d'une classe pour appliquer à celle-ci une politique de persistance ou de concurrence. Cette pratique, typique de la méthodologie

d'*OpenC++*, est réalisée grâce à la méta-classe WrapperClass que nous spécialisons, adaptons puis appliquons à la classe souhaitée. La cohérence de l'effet désiré reste bien entendu à la charge du méta-programmeur qui doit par exemple s'assurer qu'il a bien appelé explicitement la méthode originelle dans son *wrapper*.

2.3.4 *OpenJava*

OpenJava [25, 97] constitue un système méta-objet pour *Java*, basé sur des macros. La démarche est relativement similaire à celle que nous venons de décrire pour *OpenC++* : chaque entité du programme écrit en *OpenJava* est représentée par un méta-objet dont la principale fonction est la traduction de ses instances en *Java* standard.

Ici la classe racine se nomme OJClass et elle se trouve donc être la méta-classe par défaut de toute classe *Java*. En effet, sa tâche consiste simplement à reporter le code *OpenJava* vers du *Java* sans modification. Pour enrichir ou modifier le comportement d'une classe, il faut donc spécialiser OJClass, redéfinir certaines de ces méthodes — telles translateDefinition dont le rôle est justement la traduction d'*OpenJava* vers *Java* du code d'une classe — puis lancer la traduction. Par la suite, une compilateur ou interprète *Java* habituel fera tout à fait l'affaire.

De manière à peu près similaire à *OpenC++*, il est possible de modifier le comportement de l'héritage et de l'agrégation. Cependant, *OpenJava* reste nettement ciblé sur la notion de classe, et beaucoup moins sur les relations entre elles qui constitue le cœur de nos préoccupations et justifie notre approche. La marque la plus flagrante en est la prépondérance de la méta-classe OJClass sur les autres méta-objets. Ainsi, la création d'une nouvelle relation n'est absolument pas prévue. Évidemment, comme dans tout système, rien n'empêche toutefois de la réaliser complètement à la main.

2.3.5 *Javassist*

L'approche suivie par *Javassist* [24, 98] est relativement différente de celles que nous venons de présenter. *Javassist* se présente comme un assistant mettant à la disposition du programmeur un système d'aide à la programmation. Il repose cependant sur une architecture méta, comme ces prédécesseurs dans cet état de l'art.

Une spécificité de *Javassist* par rapport à *OpenJava* est de n'être pas un simple traducteur vers *Java*. *Javassist* traduit quelques particularités syntaxiques vers du *Java* standard mais génère également directement du *byte-code* pour la machine virtuelle lorsque cela s'avère utile. Une édition de liens concrétise ensuite le tout. Autre différence, *Javassist* agit certes à la compilation, en modifiant le code initial, mais beaucoup plus à l'exécution par ajout automatique de classes qui seront utilisées par l'application. En fait, cette dernière capacité est le principal point fort de *Javassist*. Il est par exemple possible de générer, avec un travail limité, une classe simulant un container générique contraint. Par exemple, créer un vecteur de personnes[15] ne nécessitera pas de réécrire une fastidieuse classe VectorOfPersonnes. En effet, une telle requête est prévue en *Javassist* : elle génère cette nouvelle classe et l'intègre à celles présentes dans la machine virtuelle pour permettre son utilisation.

Pour résumer la philosophie de *Javassist*, nous dirons qu'il est conçu pour produire facilement une nouvelle classe sur une requête plutôt que pour adapter le comportement d'une classe existante, ce qui est souvent l'apanage des systèmes méta-

[15] Une *personne* signifie ici une instance de la classe Personne.

objets. Faciliter la génération d'une classe, voilà qui met de nouveau la classe au centre du système, *Javassist* reste donc également mal outillé pour l'adaptation ou la création de relations entre classes.

2.3.6 *metaXa* (*MetaJava*)

metaXa [71, 59] s'est initialement appelé *MetaJava*. *metaXa* constitue un niveau méta de réification du système (principalement un programme *Java* bien que l'approche puisse être généralisée) décrit par le programmeur. Ce niveau méta est activé par des événements déclenchés par le programme lui-même. Par exemple, à chaque appel de méthode, un événement active le niveau méta. Celui-ci prend le contrôle total du programme —qui est donc suspendu —, réalise les tâches utiles, puis réactive l'exécution du programme là où elle en était restée. L'accès à une variable, la création d'un objet ou le chargement d'une classe sont également des événements déclenchés au niveau du langage pour donner la main au niveau méta.

La réification réalisée par *metaXa* n'est pas exhaustive pour rester performante : seules les entités indispensables sont modélisées au niveau méta. Cela permet de conserver un implantation au plus juste, sans travail inutile.

Par un choix arbitraire mais bien ciblé des moments sensibles — tels l'appel d'une primitive — de l'exécution d'un programme, les concepteurs de *metaXa* ont ouvert cette exécution à des adaptations réalisées par un protocole méta-objet activé grâce à un système événementiel. Cette approche, relativement originale, permet effectivement de modifier le comportement du langage sous-jacent. Cependant, pour que le système reste utilisable, le nombre de points d'entrée est forcément limité. Ainsi, il n'est pas envisageable de multiplier ceux-ci pour modifier facilement un comportement non modélisé, tel l'héritage.

Le projet *metaXa* est aujourd'hui clos. Les avancées réalisées par *metaXa* sont désormais mises à profit dans un nouveau projet d'architecture flexible de systèmes d'exploitation : *JX* [60].

2.3.7 *Iguana*

Iguana [62, 51] est un protocole méta-objet pour *C++*[16]. Il est défini au travers d'une extension de la syntaxe de *C++* et agit tel un pré-processeur. *Iguana* lit le code source étendu, interprète et exécute les modifications apportées par le niveau méta, puis produit le programme en *C++* en l'équipant des extensions utiles.

Iguana dispose d'un mécanisme de réification sélectif. Le méta-programmeur indique ainsi au système quelles sont les entités qui doivent être réifiées et à quelle méta-entité elles seront liés. Ainsi, il est par exemple possible d'associer à un objet une politique d'envoi de message particulière en lui spécifiant `reify Dispatch : MyDispatcher`. `reify` indique de produire un objet méta associé, `Dispatch` signale quelle caractéristique est concernée (ici, l'envoi de message) et `MyDispatcher` signale la méta-classe qui décrit la politique d'envoi de message à appliquer.

Les capacités d'adaptation d'*Iguana* reposent principalement sur les notions de classe, de primitive (méthode ou attribut) et de message. Ainsi, il est par exemple possible de modifier le comportement de la liaison dynamique en adaptant la manière donc les méthodes sont appelées. Cependant, aucune structure ne permet d'ajouter facilement un type de relation entre classes au langage décrit.

[16] Une version d'*Iguana* existe aussi pour *Java*.

2.3.8 *FLO*

Nous souhaitons enfin aborder le langage *FLO* [53, 52]. Ce langage repose sur un modèle qui a retenu tout notre intérêt en raison de la présence de méta-objets pour les relations.

FLO est en effet une implémentation qui utilise les caractéristiques réflexives de *CLOS* pour assurer le maintien de la cohérence de dépendances entre objets. Les dépendances elles-mêmes sont réifiées et disposent donc d'un méta-objet correspondant, ce qui se trouve être parfaitement dans le fil de notre approche.

FLO dispose d'un modèle avec un *bootstrap* inspiré de ceux d'*ObjVlisp* ou *CLOS-MOP*, qui assure que toutes les entités de *FLO* sont elles-mêmes décrites en *FLO* et qui offre au modèle puissance et élégance.

Plusieurs différences nous ont cependant un peu éloignés de *FLO*. Tout d'abord un aspect historique : *FLO* repose sur des langages très dynamiques comme peuvent l'être *CLOS* ou *Smalltalk* alors que la genèse de notre étude faisait plutôt cas d'*Eiffel* et de *Java*. De plus, les relations réifiées par *FLO* le sont dans le but de gérer un modèle de dépendances (le terme de *dépendance* est d'ailleurs préféré à *relation* dans *FLO*) entre objets là où nous nous intéressons plutôt aux relations inter-classes, qui définissent et régissent des relations inter-objets.

2.3.9 Bilan

Nous venons de résumer les fonctionnalités et les choix de plusieurs protocoles méta-objets que nous avons sélectionnés. Il en existe d'autres ; rien que pour *Java*, citons par exemple *Guaraná* [84], *KAVA* [101] (anciennement *Dalang* [100]) et *ProActive* [19]. Avant de passer à une autre section de cet état de l'art, nous aimerions présenter un bilan concernant les protocoles méta-objets étudiés.

Tout d'abord, remarquons que la totalité de ces systèmes, à l'exception d'*OpenC++* et d'*OpenJava*, agissent, au moins en partie, durant l'exécution. *OpenC++* et *OpenJava* réalisent quant à eux tout le travail du méta niveau à la compilation. Cette dernière solution présente l'avantage de minimiser les pertes de performance en temps et en mémoire pendant le déroulement du programme. Si cette approche n'est pas la plus utilisée, c'est que l'action pendant l'exécution (une partie du travail, mais une partie seulement, est réalisée à la compilation) permet une structuration méta plus complète — car la réification subsiste — et, parfois, une modification dynamique du comportement.

La plupart de ces protocoles méta-objets offrent des points d'entrée à des moments clés de l'exécution du programme modélisé. Nous retrouvons notamment parmi ces derniers l'appel d'une méthode, l'appel d'un constructeur, l'accès à un attribut ou la création d'un objet. Il est également parfois possible d'intervenir lors du chargement d'une classe. Notre préoccupation est tournée vers les relations entre classes. Par un détournement de l'appel de méthode, il est toujours possible de méta-programmer un comportement particulier simulant une relation spécifique. Cependant cette technique reste particulièrement lourde à mettre en œuvre car il est alors nécessaire de réécrire bon nombre d'algorithmes du langage (structure du schéma de types, liaison dynamique, masquage, redéfinition, ...).

La classe est et reste la structure de base à méta-modéliser pour tous ces protocoles méta-objets, l'appel de méthode étant le moment privilégié où les algorithmes de niveau méta sont activés. Nous pensons, pour notre part, que si la classe est importante, les relations entre classes le sont encore plus. Aussi le modèle *OFL* que nous

entamons de décrire à partir du chapitre 3 page 47 met-il ces relations au centre de son système. Mais avant d'en arriver là, nous souhaitons vous donner un rapide aperçu de quelques approches qui ne sont pas des protocoles méta-objets mais qui présentent un intérêt certain pour notre démarche.

2.4 Autres systèmes

Cette dernière section de notre état de l'art est rédigée pour présenter des systèmes, modèles, méthodes ou outils dont la philosophie ou l'usage ont inspirés notre étude. Nous faisons donc ici un rapide survol de ces technologies.

2.4.1 *UML*

Premiers des éléments connexes que nous souhaitons évoquer, *UML* (*Unified Modeling Language*) [73, 5, 90, 67, 83] est un langage et une méthode d'analyse et de conception de systèmes informatiques. Notre intérêt à son endroit repose sur trois idées :

1. Le formalisme graphique défini par *UML* (les conventions pour dessiner une classe, un objet, un héritage, ...) est utilisé et compris par l'ensemble de la communauté *objets*, voire par l'ensemble des analystes-programmeurs. Ce formalisme est donc une base essentielle à ne pas négliger.
2. *UML* ne limite pas les relations entre classes disponibles à celles présentes dans les langages de programmation qui seront utilisés pour l'implémentation à suivre. Nous retrouvons par exemple la relation d'*association* qui peut-être mise en œuvre par un patron de conception, par exemple. Il est également possible, en *UML* de signaler un type de relation spécifique simplement en nommant la flèche qui unit les classes entre elles. En laissant au soin du programmeur de s'arranger avec cela par la suite...
3. Enfin, *UML* a peu à peu supplanté, notamment dans les domaines couverts par les technologies objets, toutes les autres méthodes de conception. La capacité d'un projet logiciel tel le nôtre à être diffusé et compris dépend aussi du choix d'un standard reconnu par une communauté, à défaut d'être universel.

2.4.2 *XML*

XML (*eXtensible Markup Language*) [80, 102] est un langage de balises, successeur de *SGML* (*Standard Generalized Markup Language* [2]) et compatible avec ce dernier, dont la fonctionnalité est de décrire dans un format simple, standard et extensible un ensemble de données. Le but d'un tel langage est de fournir une base commune simple à tous les interlocuteurs des réseaux informatiques pour permettre un échange de données facilitée. *XML* n'a donc pas vocation à être utilisé comme format de base pour chaque application mais plutôt comme support d'exportation et d'importation rendant les données indépendantes des programmes qui les traitent.

A priori, tout cela n'a pas grand-chose à voir avec notre problématique (qui est de réaliser, rappelons-le, un modèle méta-objet centré sur les relations inter-classes). En fait, *XML* y est moins étranger qu'il n'y paraît. En effet :

1. Le temps des outils monolithiques est désormais révolu en informatique, sauf peut-être pour des applications très réduites ou des domaines très spécifiques.

Aujourd'hui, un système logiciel est souvent formé de plusieurs entités qui communiquent ensemble pour concourir au résultat attendu. Et qui dit communication dit langage commun. Ici deux écoles s'affrontent. La première préfère utiliser des formats d'échange propriétaires. Ceux-ci ont l'avantage d'être parfaitement adaptés à leur usage et au contexte, puisqu'ils ne sont fait que pour cela. Il est également permis de leur associer une qualité commerciale qui est d'imposer un outil pour la simple raison qu'il est le seul à comprendre les résultats d'un autre. La seconde solution consiste à choisir un format standard tel *XML*. L'inconvénient est que le format n'est pas forcément parfaitement adapté à l'usage qu'on attend de lui et qu'il faut donc alors faire preuve de plus d'initiative. D'un autre côté, chaque outil s'avère alors interchangeable puisque reposant sur un format ouvert. De plus, *XML* a la particularité d'être lisible avec un logiciel aussi simple qu'un éditeur de texte, ce qui rend son usage particulièrement aisé.

2. *XML* décrit des données et non des programmes ou des algorithmes. Cela semble une restriction importante. En fait, il n'en est rien. En effet, au niveau méta, les programmes et les algorithmes ne sont finalement que des données.

Signalons aussi l'existence de *XMI* (*XML Metadata Interchange*) [81], système dont le but est de permettre l'échange de méta-données *XML* entre des outils de modélisation basés sur *UML* (cf. section précédente) et des bases de méta-données reposant sur *MOF* (cf. section suivante).

2.4.3 *MOF*

Troisième et avant-dernier sujet connexe que nous souhaitons aborder, *MOF* (*Meta-Object Facility*) [37, 82] est un format de description de méta-modèles, en quelque sorte un méta-méta-modèle. *MOF* apporte principalement deux avantages :

1. Il est un moyen standard de décrire des capacités et structures de niveau méta. En cela, il permet de reposer sur un formalisme reconnu et adapté à la description de méta-modèle.
2. Il garantit que tous les méta-modèles qui l'utilisent comme format de description peuvent stocker leur données au sein d'une base commune. *MOF* améliore donc également l'interopérabilité entre méta-modèles.

L'importance que nous accordons à *MOF* est celle qui est due à un format standard. Notre démarche[17] nous conduit à décrire un modèle méta-objet et nous pensons évidemment que *MOF* constituera, même s'il ne s'agit pas d'un objectif à court terme, un support important pour celui-ci. En d'autres mots, nous considérons à terme comme un but de décrire notre modèle à l'aide de *MOF* pour lui assurer une meilleure formalisation et une lisibilité accrue.

2.4.4 Patrons de conception

Nous finirons cet état de l'art en disant quelques mots à propos des patrons de conception (*design patterns*) [56]. En effet, notre travail repose principalement sur l'idée que le nombre de sortes de relations entre classes est trop faible dans les langages à objets actuels[18]. Ainsi, ceux qui sont présents sont mis à contribution pour réaliser des tâches très différentes.

[17] Patience ! Plus qu'une toute petite sous-section et nous y serons...

[18] Ce qui ne signifie pas qu'il faille en ajouter beaucoup, ce qui serait à l'évidence contre-productif.

Les patrons de conception sont un moyen particulièrement efficace et élégant de pallier cette carence. S'appuyant presque exclusivement sur l'héritage et l'agrégation, ils proposent un panoplie de solutions applicables à un ensemble de problèmes donnés. Ces problèmes sont ceux que l'on retrouve fréquemment lors de la phase d'implémentation d'une application. Autrement dit, un patron de conception est formé d'un agencement d'éléments — classes et objets reliés par des héritages, agrégations et instanciations — qui permet de résoudre un problème de conception logicielle fréquent.

À notre sens, les patrons de conception sont et resteront une des techniques parmi les plus importantes en matière de programmation. Notre approche consiste à compléter, et non à remplacer, cette idée en mettant à la disposition du programmeur un ensemble de relations plus étoffé qui, pourquoi pas, feront elles aussi partie de futurs patrons ce conception.

2.5 Bilan

Dans ce chapitre, nous avons tout d'abord étudié relativement en détail les deux langages de programmation qui nous ont principalement servis de base. Puis, nous avons abordé le thème des protocoles méta-objets qui constituent la voie que nous avons choisie pour apporter notre contribution. Enfin, nous avons évoqué quelques sujets connexes qui ont enrichi notre réflexion et nous ont servis d'outils ou de sources d'inspiration.

Chapitre 3

OFL : étude préliminaire

3.1 Introduction

Pour débuter notre présentation du modèle *OFL* (rappelons qu'*OFL* signifie *Open Flexible Languages*), nous proposons une étude ciblée des langages *Eiffel* et *Java*. Nous décrivons d'abord rapidement les notions fondamentales du modèle : *composant-langage* (cf. définition 1), *composant-description* (cf. définition 2) et *composant-relation* (cf. définition 4).

Définition 1 (composant-langage) *Un composant-langage est un composant logiciel qui offre une définition d'un langage de programmation et, dans le contexte de notre étude plus précisément, d'un langage à objets à classes.* En d'autres termes, un composant-langage est un méta-langage.

Définition 2 (composant-description) *Un composant-description est un composant logiciel qui offre une définition d'une description (cf. définition 3) dans un langage à objets à classes.* Un composant-description est donc une méta-classe.

Définition 3 (description) *Une description est une entité qui offre une définition d'une instance.* Les *classes* sont les descriptions les plus connues des langages à objets. Mais les interfaces ou les types de tableau de *Java* sont aussi, par exemple, des descriptions.

Définition 4 (composant-relation) *Un composant-relation est un composant logiciel qui offre une définition d'une relation (cf. définition 5) dans un langage à objets à classes.* En ce sens, un composant-relation s'apparente à une méta-relation[1].

Définition 5 (relation) *Une relation est une entité d'une application qui assure la liaison entre d'autres entités, que ces dernières soient des descriptions (cf. définition 3) ou des instances finales.* L'héritage, l'agrégation ou l'instanciation sont des relations communes dans les langages à objets à classes.

Pour simplifier les figures suivantes, nous donnons dans la figure 3.1 page suivante la légende générale qui, sauf mention explicite contraire, s'applique à toutes. Dans cette figure, on peut considérer en première approche que le mot *description* peut être remplacé par le mot *classe*. Pour le reste, le formalisme utilisé est largement inspiré de celui d'*UML*.

[1]Le mot relation est, en français, aussi bien employé pour indiquer une liaison de manière générale que *la* liaison particulière qui concrétise la liaison générale sur des valeurs définies. Nous avons besoin de lever cette ambiguïté et choisissons de nommer *type de relation*, *méta-relation* ou *composant-relation* la première acception, et *relation* la seconde.

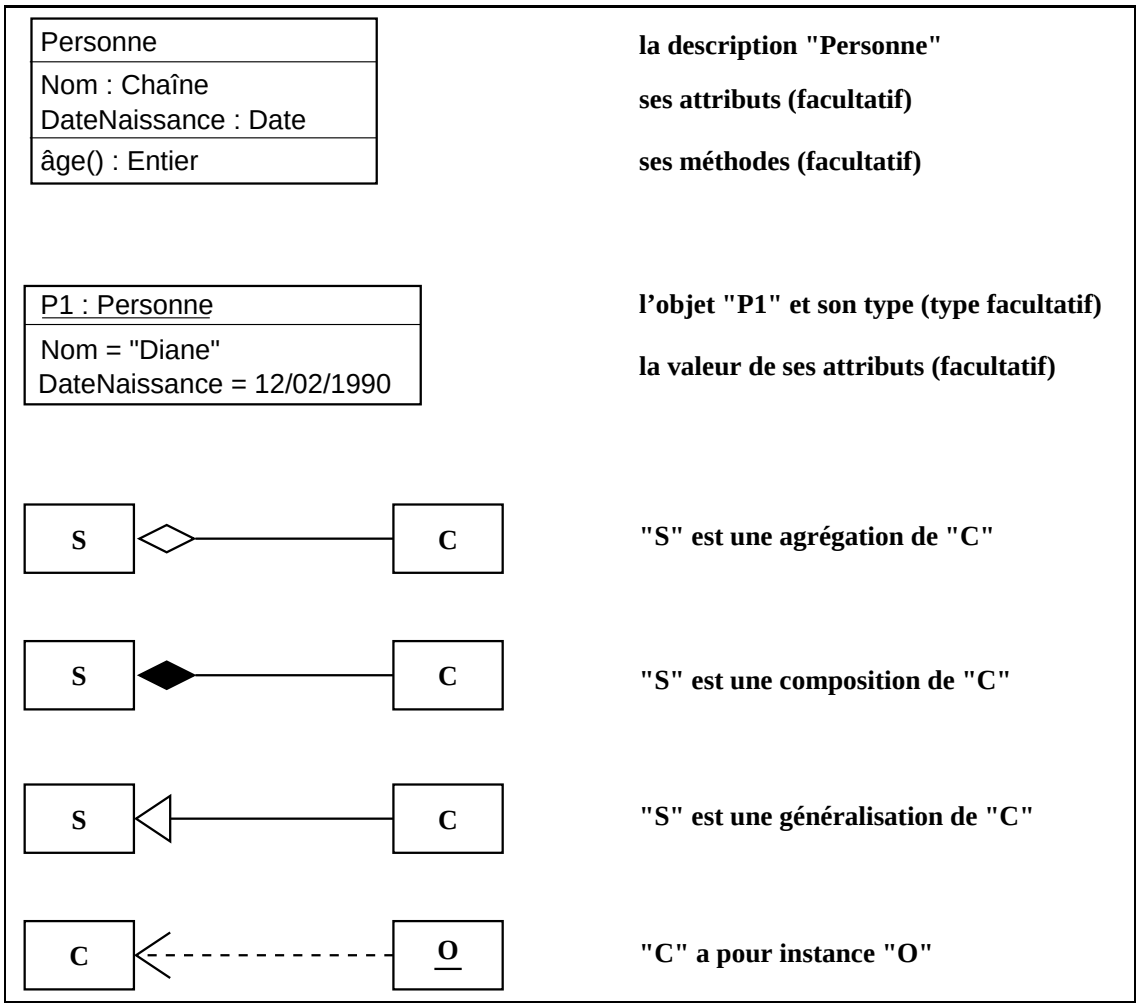

FIG. 3.1 – Légende générale pour *OFL*

Cette section s'attache à introduire les trois notions fondamentales ci-dessus, à présenter des exemples tirés des langages de programmation *Eiffel* et *Java* et à décrire la sémantique opérationnelle de ces langages.

3.2 Notion de composant-relation

3.2.1 Présentation

La première notion fondamentale du modèle est constituée par les composants-relations. Nous nous sommes particulièrement intéressés aux relations inter-descriptions (la notion de classe est généralisée par le terme *description*), et principalement à l'héritage qui a retenu toute notre attention.

Pour chaque relation, nous avons défini les notions de *description-source* (cf. définition 6) et *description-cible* (cf. définition 7).

Définition 6 (description-source) *Une description-source d'une relation (cf. définition 5 page 47) inter-description est une description (cf. définition 3 page 47) qui déclare cette relation et qui indique ainsi vouloir bénéficier du service correspondant.* Par exemple, dans un héritage la description-source est l'héritière, dans une agrégation c'est la description qui déclare un attribut.

Définition 7 (description-cible) *Une description-cible d'une relation (cf. définition 5 page 47) inter-description est une description (cf. définition 3 page 47) qui fournit un service demandé par une description-source (cf. définition 6).* Par exemple, dans un héritage la description-cible est l'héritée, dans une agrégation c'est la description qui définit le type d'un attribut.

Pour donner une autre exemple, citons la relation implements de *Java* : la description-source est la classe qui concrétise un ensemble de descriptions-cibles constitué par les interfaces implémentées. À moins que le contraire ne soit précisé, une source (respectivement : cible) peut être indifféremment directe ou indirecte. Ce fait est illustré sur la figure 3.2[2].

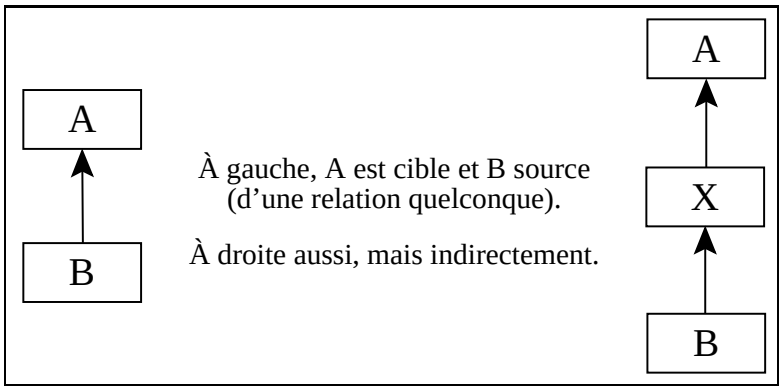

FIG. 3.2 – Relation directe ou indirecte

Le modèle que nous présenterons décrit les relations inter-descriptions (exemples : héritage, agrégation), les relations entre descriptions et objets (exemples : instanciation, extension) et peut aussi définir des relations inter-objets, mais ces dernières ne font pas partie du domaine de recherche défini pour cette thèse.

[2]Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page précédente.

Prenons deux exemples pour appuyer notre explication : *Eiffel* et *Java*. Pour chacun des composants-relations nous donnerons, sauf dans les cas trop atypiques, les propriétés suivantes :

- La relation constitue-t-elle une importation ou une utilisation des primitives de la description-cible par la description-source ? Une importation signifie que les primitives de la cible sont, intégralement ou en partie, à l'identique ou avec modification, reproduites dans la source. Nous parlons d'utilisation lorsque la source fait appel à une service de la cible sans pour autant l'importer.
- La relation est-elle simple ou multiple (dans ce cas, elle peut admettre plusieurs descriptions-cibles) ?
- La relation peut-elle être répétée (dans ce cas, elle peut admettre que la même description apparaisse plusieurs fois comme description-cible) ? Si elle est simple, elle est forcément non répétée.
- La relation est-elle obligatoirement linéaire ou admet-elle des cycles ?
- Les primitives de la description-cible sont-elles accessibles directement (comme si elles étaient locales) ou indirectement (par un nommage explicite de la description-cible ou de l'une de ses instances) par la description-source ? Cette question est illustrée sur la figure 3.3[3].

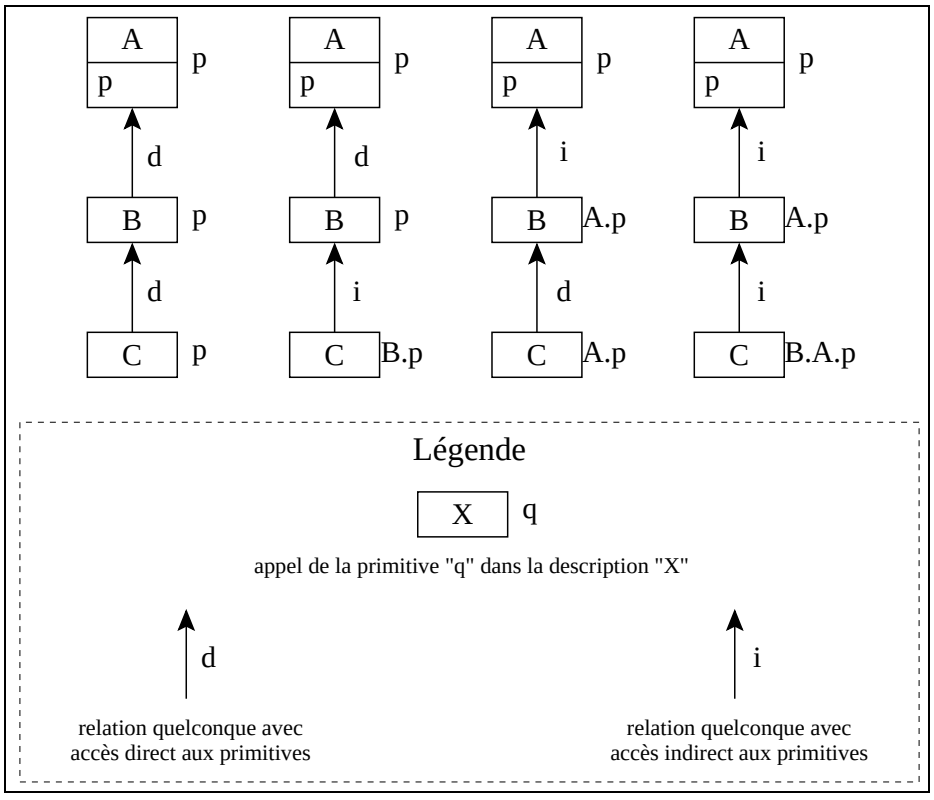

FIG. 3.3 – Accès direct ou indirect

- Cette relation implique-t-elle un polymorphisme (cf. définition 8 page ci-contre) pour les instances des descriptions liées ? Cette question ne prend de sens que dans le cas d'une relation d'importation.
- Toujours exclusivement pour les relations d'importation, quel type de variance (covariance, contravariance, ...) est acceptée ?

[3]Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48.

- Pour les relations d'utilisation seulement, l'objet utilisé est-il dépendant ou indépendant de celui qui l'utilise. Nous parlons d'objet dépendant quand la *vie* de l'utilisé est conditionnée par celle de l'utilisant. C'est par exemple le cas quand l'objet utilisé, plutôt qu'une référence, est une simple valeur.
- Nous nous posons ensuite la question de savoir si l'objet de la description-cible est utilisé systématiquement, au travers de la relation, par tous les objets de la description-source (notion d'*attribut de description*) ou par un ou plusieurs objets de la description-source (*attribut d'instance*) ou encore par un unique objet de la description-source (comme les attributs expansés d'*Eiffel*). Cette question ne prend de sens, comme la précédente, que dans le cas d'une relation d'utilisation.
- Les deux questions suivantes, spécifiques aux relations d'utilisation elles-aussi, concernent la possibilité d'accéder directement aux attributs ou le fait de devoir obligatoirement passer par un accesseur. La question se pose en lecture puis en écriture (modification).
- Il est ensuite question de symétrie. La relation est-elle ou non symétrique ? C'est-à-dire fournit elle un lien identique de la source vers la cible et de la cible vers la source ?
- Nous cherchons ensuite à reconnaître une éventuelle relation opposée.
- Nous voulons ensuite déterminer quelles sont les opérations permises par le concept-relation parmi :
  - supprimer une primitive,
  - renommer une primitive,
  - redéfinir une primitive,
  - masquer une primitive,
  - montrer (*dé*masquer) une primitive,
  - rendre une primitive abstraite et
  - rendre une primitive concrète.
- Enfin, nous donnons la liste des composants-descriptions (que nous décrivons section 3.3.2 page 61) qui peuvent servir de source à ce composant-relation, puis celle de ceux qui peuvent être utilisés comme cible.

Définition 8 (polymorphisme) *Le polymorphisme est la capacité d'un objet à être utilisé selon la définition donnée dans plusieurs descriptions (cf. définition 3 page 47). Nous considérons ici un polymorphisme d'inclusion qui indique que l'extension d'une description est constituée de l'union de ses instances propres et des extensions des descriptions qui la spécialisent.*

3.2.2 Les composants-relations d'*Eiffel*

Le langage *Eiffel* définit plusieurs composants-relations inter-descriptions : un composant-relation d'importation : l'héritage et six composants-relations d'utilisation : la clientèle, la clientèle expansée, la clientèle générique, la clientèle générique expansée, la généricité et la généricité expansée (cf. sections 2.2.1 page 23).

héritage (mot-clé inherit) Prenons un exemple dans lequel une classe S (pour *Source*) hérite d'une classe C (pour *Cible*), S et C pouvant être concrètes ou abstraites (avec, dans ce cas, les restrictions d'usage sur la gestion des instances propres). Si nous essayons de résumer les caractéristiques de l'*héritage à la Eiffel*, nous pouvons dire que ce composant-relation est :
- importation,

- multiple (S peut hériter d'autres classes),
- répété (S peut hériter plusieurs fois, directement ou indirectement, de C),
- linéaire (C ne peut hériter de S, ni directement ni indirectement),
- accès directs permis (conséquence de l'importation des primitives) et indirects non applicables (il n'existe aucun moyen d'appeler une primitive de la classe-cible par l'intermédiaire de la relation d'héritage),
- implique un polymorphisme (chaque instance de S est aussi une instance de C et peut-être utilisée comme telle),
- covariant (dans S, tous les attributs, résultats de fonction et paramètres de méthode[4] doivent être d'un type conforme [et de même nombre, pour les paramètres] à leur homologue dans C [cf. section 2.2.1 page 23 pour la définition de la conformité de type]),
- asymétrique (« S hérite de C » n'implique pas « C hérite de S »),
- sans opposé (l'héritage permet de spécifier, comme description-cible, une classe ascendante mais aucune relation n'existe pour faire l'inverse : déclarer une descendante),
- l'ajout, le renommage, la redéfinition, l'abstraction et la concrétisation sont autorisés (sous certaines conditions, par exemple pour la redéfinition : même nombre de paramètres, conformité des types de paramètres [cf. section 2.2.1 page 23], ...), la suppression, le masquage et le démasquage sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe, classe abstraite, classe générique, classe abstraite générique, classe expansée, classe expansée générique et type fondamental (cf. page 64).

clientèle Imaginons une classe S qui possède un attribut (référencé) de classe C. Cette relation possède les caractéristiques suivantes (le polymorphisme et la variance ne sont pas applicables aux composants-relations d'utilisation, au contraire de la dépendance et du partage qui leur sont spécifiques). Cette relation peut être décrite par les caractéristiques suivantes :
- utilisation,
- multiple[5] (S peut avoir d'autres attributs),
- répété (S peut avoir d'autres attributs de classe C),
- circulaire (directement : S peut posséder un attribut de classe S, ou indirectement : C peut posséder un attribut de classe S),
- accès directs non applicables (il est indispensable de passer par l'attribut pour appeler une de ses primitives) et indirects permis (pour toutes les primitives non masquées, l'accès se fait par l'intermédiaire de l'attribut),
- indépendant (l'objet de classe C attaché à l'attribut a une durée de vie indépendante de l'objet de classe S qui l'utilise),

[4] Désormais, pour simplifier la lecture, sans que cela nuise cependant à la généralité, nous parlerons souvent des attributs seulement et négligerons les résultats de fonction, paramètres de méthode, ...

[5] On peut considérer soit que la relation de clientèle est multiple, soit qu'elle peut être composée avec elle-même, ce qui revient au même.

- attribut d'instance[6] (plusieurs instances de S peuvent référencer la même instance C),
- attributs directement accessibles en lecture (des limitations peuvent être introduites par les clauses d'exportation),
- accesseurs obligatoires pour la modification des attributs (d'autres limitations peuvent être introduites par les clauses d'exportation),
- asymétrique (« S est cliente de C » n'implique pas « C est cliente de S »),
- sans opposé (aucune relation ne permet à une classe de choisir d'avoir un attribut dans une autre),
- le masquage et le démasquage sont autorisés, l'ajout, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe, classe abstraite, classe générique, classe abstraite générique et néant (cf. page 65).

clientèle expansée Prenons maintenant une classe S qui possède un attribut expansé de classe C ou un attribut dont la classe C est expansée. Cette nouvelle forme de clientèle est définie par les propriétés suivantes :

- utilisation,
- multiple,
- répété,
- linéaire (C ne peut posséder un attribut expansé de classe S),
- accès directs non applicables et indirects permis (sauf pour les primitives masquées),
- dépendant (l'objet de classe C attaché à l'attribut expansé a une durée de vie dépendante de l'objet de classe S qui l'utilise),
- attribut d'instance unique[7] (plusieurs instances de S ne peuvent pas référencer la même instance de C),
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- asymétrique,
- sans opposé,
- le masquage et le démasquage sont autorisés, l'ajout, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe, classe générique, classe expansée, classe expansée générique, type fondamental (cf. page 64) et type binaire (cf. page 64).

[6] En utilisant une fonction once, il est possible de créer l'équivalent d'un attribut de description.
[7] Une fonction once permet de gérer un attribut de description unique.

clientèle générique Une classe S qui possède un attribut générique de classe C. Cette nouvelle forme de clientèle est définie par les propriétés suivantes :
- utilisation,
- multiple,
- répété,
- circulaire,
- accès directs non applicables et indirects permis (sauf masquage),
- indépendant,
- attribut d'instance,
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- asymétrique,
- sans opposé,
- le masquage et le démasquage sont autorisés, l'ajout, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe générique et classe expansée générique.

  La principale différence entre une clientèle et une clientèle générique — en dehors de la gestion du ou des paramètres génériques eux-mêmes — se retrouve dans la liste des cibles directes possibles. La clientèle générique ne peut s'appliquer que vers une classe générique, éventuellement expansée.

clientèle expansée générique Imaginons maintenant un attribut d'un type expansé et générique. Voici la définition de la relation correspondante :
- utilisation,
- multiple,
- répété,
- linéaire,
- accès directs non applicables et indirects permis (sauf masquage),
- dépendant,
- attribut d'instance unique,
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- asymétrique,
- sans opposé,
- le masquage et le démasquage sont autorisés, l'ajout, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe générique et classe expansée générique.

généricité Voyons comment peut être décrit la relation entre une classe S générique dont un paramètre de généricité est de classe C : C n'est connue qu'à l'exécution (cf. section 2.2.1 page 23). Ce composant-relation est :
- utilisation,
- multiple (S peut avoir plusieurs paramètres de généricité),
- répété (S peut avoir d'autres paramètres de généricité de classe C),
- circulaire,
- accès directs non applicables et indirects permis (sauf masquage),
- indépendant,
- attribut d'instance[8],
- attributs directement accessibles en lecture,
- accesseurs obligatoires pour la modification des attributs,
- asymétrique,
- sans opposé,
- le masquage et le démasquage sont autorisés, l'ajout, la suppression, le renommage, la redéfinition, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe générique (cf. page 62), classe abstraite générique (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe (cf. page 61), classe abstraite (cf. page 62), classe générique et classe abstraite générique.

généricité expansée C'est une simple composition de la clientèle expansée (cf. section 3.2.2 page 53) et de la généricité (cf. section 3.2.2 page précédente). Nous n'en décrivons donc que les parties non communes à ces deux sortes de description :
- linéaire,
- dépendant,
- attribut d'instance unique,
- composants-descriptions applicables en tant que :

  source directe : classe générique (cf. page 62), classe abstraite générique (cf. page 63) et classe expansée générique (cf. page 63) ;

  cible directe : classe expansée (cf. page 63), classe expansée générique et type fondamental (cf. page 64).

3.2.3 Les composants-relations de *Java*

Nous proposons de réaliser le même genre d'étude pour le langage *Java*. Nous trouvons quatre composants-relations d'importation (héritage inter-classe, héritage inter-interface, concrétisation et implémentation) et quatre composants-relations d'utilisation (agrégation, agrégation de classe, composition et composition de classe).

héritage inter-classe La classe S hérite (mot-clé extends) de la classe C, S et C pouvant être concrètes ou abstraites. Nous considérons ici le cas où les classes sont toutes deux concrètes ou bien où S est abstraite et C concrète. Lorsqu'une classe concrète hérite d'une classe abstraite, nous appelons cette relation *concrétisation* (cf. section 3.2.3 page 57). Ce composant-relation est :

[8] Le mot *attribut* peut être considéré ici comme un abus de langage.

- importation,
- simple (S ne peut hériter d'une autre classe),
- non répété (S ne peut hériter plusieurs fois de C),
- linéaire (C ne peut hériter de S, directement ou indirectement),
- accès directs autorisés (sauf pour les constructeurs et les primitives déclarées private) et indirects autorisés (grâce au cas particulier du mot-clé super),
- implique un polymorphisme (chaque instance de S est aussi une instance de C et peut-être utilisée comme telle),
- invariant pour la redéfinition (dans S, tous les attributs, résultats de fonction et paramètres de méthode doivent être de même type [et même nombre, pour les paramètres] que leur homologue dans C) et libre pour le remplacement (dans S, tous les attributs et paramètres de méthode doivent être de type [ou de nombre, pour les paramètres] différent de leur homologue dans C) : cf. section 2.2 page 23 pour les règles de remplacement (*hiding*), de redéfinition (*overriding*) et de surcharge (*overloading*) et pour les qualifieurs d'accès public, protected, private, static et final,
- asymétrique,
- sans opposé,
- l'ajout, la redéfinition, le masquage (par exemple en déclarant un attribut de la description-source comme private : cf. section 2.2 page 23) et l'abstraction sont autorisés (sous certaines conditions, par exemple pour la redéfinition : même nombre de paramètres, même type pour les paramètres [cf. section 2.2 page 23], ...), la suppression, le renommage, le démasquage et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69) et classe anonyme (cf. page 69) ;

  cible directe : classe, classe abstraite, classe membre statique et aussi classe membre statique abstraite, classe membre, classe membre abstraite, classe locale et classe locale abstraite.

héritage inter-interface L'interface S hérite (mot-clé extends) de l'interface C. Les caractéristiques de ce composant-relation sont :

- importation,
- multiple,
- répété,
- linéaire,
- accès directs et indirects non applicables (il n'y a pas d'instruction, pas de moyen donc d'accéder à quoi que ce soit dans la description-cible),
- implique un polymorphisme (chaque instance, forcément indirecte, de S est aussi une instance de C),
- invariant,
- asymétrique,
- sans opposé,
- l'ajout et la redéfinition sont autorisés, la suppression, le renommage, le masquage, le démasquage, l'abstraction et la concrétisation sont non applicables,

- composants-descriptions applicables en tant que :
  source directe : interface (cf. page 66) et interface membre statique (cf. page 69) ;
  cible directe : interface et interface membre statique.

concrétisation La classe S concrétise[9] (mot-clé extends) la classe abstraite C. Nous avons donc un composant-relation avec les caractéristiques :
- importation,
- simple,
- non répété,
- linéaire,
- accès directs autorisés (sauf pour les constructeurs et les primitives déclarées private) et indirects autorisés (grâce au cas particulier du mot-clé super),
- implique un polymorphisme,
- invariant pour la redéfinition et libre pour le remplacement,
- asymétrique,
- sans opposé,
- l'ajout, la redéfinition et le masquage sont autorisés (sous certains conditions), la concrétisation est obligatoire (il faut donner un corps à toutes les méthodes abstraites) et la suppression, le renommage, le démasquage et l'abstraction sont non applicables,
- composants-descriptions applicables en tant que :
  source directe : classe (cf. page 65), classe membre statique (cf. page 67), classe membre (cf. page 67), classe locale (cf. page 68) et classe anonyme (cf. page 69) ;
  cible directe : classe abstraite (cf. page 66), classe membre statique abstraite (cf. page 67), classe membre abstraite (cf. page 68) et classe locale abstraite (cf. page 69).

implémentation La classe S implémente (mot-clé implements) l'interface C ce qui donne les caractéristiques suivantes :
- importation,
- multiple,
- répété (S ne peut implémenter directement qu'une fois C, mais peut avoir, indirectement, plusieurs fois C comme super-interface par l'intermédiaire d'une relation d'héritage ; dans ce cas, il y a fusion),
- linéaire,
- accès directs et indirects non applicables (pas de moyen d'accéder à quoi que ce soit dans la description-cible puisqu'il s'agit d'une interface ne contenant que des signatures),
- implique un polymorphisme (chaque instance, forcément indirecte, de S est aussi une instance de C),
- invariant,
- asymétrique,
- sans opposé,

[9] Nous utilisons indifféremment le mot *concrétisation* pour nommer ce composant-relation de *Java* et pour indiquer le fait de donner un corps à une méthode. Les deux idées sont totalement distinctes (bien que liées) mais attention cependant à ne pas confondre.

- l'ajout, la redéfinition et la concrétisation sont autorisées, la suppression, le renommage, le masquage, le démasquage et l'abstraction sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69) et classe anonyme (cf. page 69) ;

  cible directe : interface (cf. page 66) et interface membre statique (cf. page 69).

agrégation Il s'agit ici d'un composant-relation d'utilisation et non plus d'importation, ce composant-relation est :

- utilisation,
- multiple,
- répété,
- circulaire,
- accès directs non applicables et indirects autorisés (pour les primitives déclarées public),
- indépendant,
- attribut d'instance,
- attributs directement accessibles en lecture (des limitations d'espace de nommage peuvent être introduites par les qualifieurs d'accès),
- attributs directement accessibles en écriture (des limitations d'espace de nommage peuvent être introduites par les qualifieurs d'accès),
- asymétrique,
- sans opposé,
- la suppression (en déclarant une primitive private par exemple) est autorisée, l'ajout, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), interface (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69), classe anonyme (cf. page 69) et interface membre statique[10] (cf. page 69) ;

  cible directe : classe, classe abstraite, interface, classe membre statique, classe membre statique abstraite, classe membre, classe membre abstraite, classe locale, classe locale abstraite[11], classe anonyme[12], interface

[10]Les interfaces et interfaces membres statiques ne peuvent pas déclarer d'attribut d'instance ni de variable locale. Cependant, l'agrégation est possible par les paramètres de méthode et résultats de fonction. La différenciation des relations d'utilisation des attributs, variables locales, paramètres de méthode et résultats de fonction n'est pas l'objet de cette thèse qui se concentre sur les relations d'importation. Elle constitue cependant une perspective bien utile de ce travail.

[11]La déclaration d'une variable locale de type d'une classe locale ou locale abstraite s'apparente en effet à une agrégation.

[12]L'usage, même éphémère, d'une instance de classe anonyme est aussi une sorte d'agrégation.

membre statique et tableau (cf. page 70).

agrégation de classe Dans ce cas, le qualifieur d'accès static est précisé. Cela signifie que l'attribut de classe C est associé à la classe S plutôt qu'à ses instances (cf. section 2.2.2 page 30). Ce nouveau composant-relation est :
- utilisation,
- multiple,
- répété,
- circulaire,
- accès directs non applicables et indirects autorisés (pour les primitives déclarées public),
- indépendant,
- attribut de description[13],
- attributs directement accessibles en lecture,
- attributs directement accessibles en écriture,
- asymétrique,
- sans opposé,
- la suppression (en déclarant un attribut private par exemple) est autorisée, l'ajout, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), interface (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69), classe anonyme (cf. page 69) et interface membre statique[14] (cf. page 69) ;

  cible directe : classe, classe abstraite, interface, classe membre statique, classe membre statique abstraite, classe membre, classe membre abstraite, classe locale, classe locale abstraite[15], classe anonyme[16], interface membre statique et tableau (cf. page 70).

composition Ce composant-relation décrit l'utilisation d'attribut de type primitif (exemples : entiers, booléens) dans la classe source. Les caractéristiques d'un tel composant-relation peuvent être décrites par :
- utilisation,
- multiple,
- répété,
- linéaire,
- accès directs et indirects non applicables (le type primitif qui sert de cible n'a aucune primitive, ce n'est pas une classe),

[13] Cette capacité à être partagé se rapproche un peu des fonctions once d'*Eiffel* mais diffère d'elles par le fait que la valeur partagée est variable.

[14] Les interfaces, classes membres, classes membres abstraites, classes locales, classes locales abstraites, classes anonymes et interfaces membres statiques peuvent déclarer une agrégation de classe pour une constante.

[15] Il est possible de faire une agrégation de classe pour une constante d'une classe locale ou locale abstraite dans une autre classe locale ou locale abstraite.

[16] Une instance de classe anonyme peut être utilisée pour initialiser une variable de classe.

- dépendant,
- attribut d'instance,
- pas d'attribut, donc pas de problème d'accessibilité,
- asymétrique,
- sans opposé,
- la suppression est autorisée, l'ajout, le renommage, la redéfinition, le masquage, le démasquage, l'abstraction et la concrétisation sont non applicables,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), interface (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69), classe anonyme (cf. page 69) et interface membre statique[17] (cf. page 69) ;

  cible directe : type primitif (cf. page 70).

composition de classe Elle peut être vue comme un mélange d'agrégation de classe (cf. section 3.2.3 page précédente) et de composition (cf. section 3.2.3 page précédente). En voici les caractéristiques remarquables :

- linéaire,
- accès directs et indirects non applicables,
- dépendant,
- attribut de description,
- pas d'attribut, donc pas de problème d'accessibilité,
- composants-descriptions applicables en tant que :

  source directe : classe (cf. page 65), classe abstraite (cf. page 66), interface (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69), classe anonyme (cf. page 69) et interface membre statique[18](cf. page 69) ;

  cible directe : type primitif (cf. page 70).

3.3 Notion de composant-description

3.3.1 Présentation

Nous venons de présenter plusieurs exemples de composants-relations tirés des langages *Eiffel* et *Java*. La présente partie s'attache à réaliser une étude similaire des descriptions de ces deux langages.

Voici les points que nous avons considérés comme essentiels dans la présentation de ces composants-descriptions :

[17] Les interfaces et interfaces membres statiques ne peuvent pas déclarer d'attribut d'instance ni de variable locale. Cependant, la composition est possible par les paramètres de méthodes et résultat de fonctions.

[18] Les interfaces, classes membres, classes membres abstraites, classes locales, classes locales abstraites, classes anonymes et interfaces membres statiques peuvent déclarer une agrégation de classe pour une constante.

- S'agit-il d'un composant-description simple c'est-à-dire associé à un seul type, ou multiple, associé à plusieurs types[19]. ?
- Ce composant-description a-t-il la capacité de créer des instances propres ?
- A-t-il la possibilité de les détruire ?
- Les primitives sont-elles encapsulées dans les descriptions ? Rappelons que tous les concepts de classe ne sont pas encapsulant. Par exemple, *CLOS* n'encapsule pas les méthodes.
- Les attributs et/ou les méthodes sont-ils permis ?
- La description peut-elle fournir des attributs d'instance unique, d'instance ou de description ou plusieurs de ces possibilités ?
- La description permet-elle la surcharge, c'est-à-dire la capacité de posséder plusieurs méthodes de même nom différent par le nom et/ou le type de leurs paramètres ?
- Quelle est la visibilité de cette description ? C'est-à-dire à partir d'où peut-on y accéder ?
- Enfin, pour chaque composant-description étudié, nous associons une liste des composants-relations qui peuvent être utilisés en prenant le composant-description comme source. Une seconde liste réalise la même chose pour les cibles. Les composants-relations ont également connaissance des composants-descriptions qui peuvent apparaître en tant que leurs sources ou cibles. Ces deux informations étant liés, il est envisagé de contrôler qu'elles sont cohérentes.

Ces questions, comme pour les composants-relations, nous conduisent à imaginer un certain nombre de critères qui vont nous permettre de caractériser ces composants.

3.3.2 Les composants-descriptions d'*Eiffel*

Nous décrivons dans cette partie neuf composants-descriptions pour *Eiffel*. Nous nous attacherons tout d'abord à préciser les caractéristiques des classes puis nous établirons des distinctions selon qu'elles sont abstraites, génériques ou expansées. Nous nous intéresserons ensuite à des cas particuliers : les types fondamentaux, les types binaires et néant.

classe C'est le composant-description essentiel d'*Eiffel*. Il est possible de le voir comme une description de la structure (les attributs) et du comportement (les méthodes) de ses instances. Il peut également être vu comme un conteneur de celles-ci, un peu particulier il est vrai, puisqu'il ne peut fournir son contenu. Imaginons donc, pour ce premier exemple, une classe C (non générique, non abstraite et non expansée). Nous pouvons décrire le composant-description correspondant de la manière suivante :
- simple (C est associé à un seul type),
- générateur (C se charge de la création de ses instances propres),
- non destructeur (C ne peut détruire ses instances. *Eiffel*, comme d'autres langages à objets possède un ramasse-miettes qui se charge de libérer la mémoire utilisée par les objets qui ne sont plus référencés. Il est donc inutile, et ce serait d'ailleurs dangereux, de permettre au programmeur de détruire des instances. D'autres langages, comme *C++* ne dispose pas de ramasse-miettes généraux et la destruction doit donc parfois être faite explicitement par le programmeur, à ses risques et périls.),

[19] Notion de type générique ou de *template*.

- encapsulant (tout attribut et toute méthode associée aux instances de C est décrite dans C),
- attributs et méthodes autorisés,
- peut fournir des attributs d'instance unique (expansé) ou d'instance (les attributs de description peuvent être simulés par les fonctions once),
- surcharge interdite,
- global (la classe est directement accessible par toute autre classe),
- composants-relations valides en tant que :

  source directe : héritage (cf. page 51), clientèle (cf. page 52), clientèle expansée (cf. page 53), clientèle générique (cf. page 54) et clientèle générique expansée (cf. page 54) ;

  cible directe : héritage, clientèle, clientèle expansée et aussi généricité (cf. page 54).

classe abstraite Les classes abstraites (mot-clé deferred) constituent un premier cas particulier de classe. Elles ne sont pas complètement spécifiées et sont dites *différées* ou *retardées* car leur spécification doit être précisée (concrétisée) par une autre classe. Cette incomplétude ne leur permet pas de créer des instances ; elles possèdent cependant les instances indirectes de leurs héritières non abstraites. Soit une classe abstraite C (non générique et non expansée), le composant-description classe abstraite est :

- simple,
- non générateur (C ne peut créer d'instance propre),
- non destructeur,
- encapsulant,
- attributs et méthodes (il faut au moins une méthode abstraite dans C ou dans une des ses classes héritées — cela pourra faire l'objet d'une vérification lors de la réification) autorisés,
- peut fournir des attributs d'instance unique ou d'instance,
- surcharge interdite,
- global,
- composants-relations valides en tant que :

  source directe : héritage (cf. page 51), clientèle (cf. page 52), clientèle expansée (cf. page 53), clientèle générique (cf. page 54) et clientèle générique expansée (cf. page 54) ;

  cible directe : héritage, clientèle et généricité (cf. page 54).

classe générique Une des capacités très utiles d'*Eiffel* est la possibilité de déclarer des classes génériques (cf. section 2.2 page 23). Voyons donc maintenant comment nous pouvons décrire une classe générique C (non abstraite et non expansée).

- multiple (C est associé à autant de types qu'il y a de combinaisons différentes des types des paramètres génériques effectifs lors de l'exécution de l'application),
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,

- peut fournir des attributs d'instance unique ou d'instance,
- surcharge interdite,
- global,
- composants-relations valides en tant que :

  source directe : tous.

  cible directe : héritage (cf. page 51), clientèle (cf. page 52), clientèle expansée (cf. page 53), clientèle générique (cf. page 54), clientèle générique expansée (cf. page 54) et généricité (cf. page 54).

classe abstraite générique Il s'agit d'une composition de classe abstraite (cf. section 3.3.2 page ci-contre) et de classe générique (cf. section 3.3.2 page précédente). Voici les caractéristiques particulières à cette sorte de description, nous oublierons tout ce qui est commun à classe abstraite et à classe générique :

- multiple,
- non générateur,
- attributs et méthodes (au moins une méthode abstraite) autorisés,
- composants-relations valides en tant que :

  source directe : tous.

  cible directe : héritage (cf. page 51), clientèle (cf. page 52), clientèle générique (cf. page 54), clientèle générique expansée (cf. page 54) et généricité (cf. page 54).

classe expansée Le dernier cas particulier de classe est constitué par les classes expansées (mot-clé expanded, cf. section 2.2 page 23). Elles permettent notamment de décrire élégamment les types de base d'une application. Décrivons donc une classe expansée C.

- simple,
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- n'autorise que les attributs d'instance unique,
- surcharge interdite,
- global,
- composants-relations valides en tant que :

  source directe : héritage (cf. page 51), clientèle (cf. page 52), clientèle expansée (cf. page 53), clientèle générique (cf. page 54) et clientèle générique expansée (cf. page 54) ;

  cible directe : héritage, clientèle expansée et aussi généricité expansée (cf. page 55).

classe expansée générique Elles possèdent à la fois les spécificités des classes expansée (cf. section 3.3.2) et des classes génériques (cf. section 3.3.2 page précédente) :

- simple,
- n'autorise que les attributs d'instance unique,
- composants-relations valides en tant que :

  source directe : tous.

cible directe : héritage (cf. page 51), clientèle expansée (cf. page 53), clientèle générique (cf. page 54), clientèle générique expansée (cf. page 54) et généricité expansée (cf. page 55).

type fondamental En *Eiffel*, certains types sont considérés comme fondamentaux. Ils représentent les entités de base de l'application. Les types fondamentaux (également appelés classes fondamentales) permettent de manipuler les valeurs les plus couramment utilisées : nous y trouvons ainsi les booléens, les caractères, les flottants[20] (simples et doubles) et les pointeurs. Ces classes peuvent être considérées comme normales à ceci près :

- Elles disposent d'une interface syntaxique particulière : des mots-clés leur sont associés (exemples : true) et surtout, il est possible d'utiliser directement des constantes de leur type (exemples : 12 est reconnu comme un entier).
- Chacune est expansée et hérite d'une classe non expansée sans en modifier autrement l'interface. Il est donc possible d'utiliser plutôt les classes héritées si nous voulons manipuler une valeur fondamentale comme un objet normal (*i. e.* référencé).
- Bien qu'il n'y ait aucune relation d'héritage entre les classes fondamentales, les entiers sont considérés comme[21] des flottants simples et les flottants simples comme des flottants doubles. Il ne s'agit cependant que de respecter une convention humaine bien pratique qui décrit l'ensemble des réels comme incluant celui des entiers.
- composants-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer une nouvelle instance du composant-description type fondamental) ;

  cible directe : héritage (cf. page 51), clientèle expansée (cf. page 53) et généricité expansée (cf. page 55).

type binaire Les types binaires sont d'un usage très spécialisé et forment un cas très particulier. Ils permettent de décrire et manipuler des objets de type *suite de n bits*[22]. Leurs particularités sont :

- Aucune classe n'existe pour décrire un objet de type *suite de n bits* avant que celui-ci ne soit déclaré. Après sa déclaration, tout se passe cependant comme si une telle classe existait et un certain nombre de primitives est donc disponible.
- Bien qu'il soit possible de s'en servir comme si elles existaient, ces classes ne sont que virtuelles et ne font donc pas partie de l'arbre d'héritage. Il est néanmoins considéré qu'elles descendent directement de la classe racine.
- Bien entendu, ces classes, qui n'existent pas, n'héritent de rien sauf de la classe racine ANY. Cependant, dans un souci de les rendre pratiques à utiliser, il est possible de considérer tout objet de type *suite de n bits* comme un objet quelconque (de la classe racine donc) ou comme un objet de type *suite de m bits*, à la condition *sine qua none* que $n \leq m$.
- composants-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer une nouvelle

[20] Les flottants sont des approximations de réels.

[21] « considéré comme » se dit « conforme à » dans la terminologie *Eiffel* (cf. section 2.2.1 page 23).

[22] n ne peut varier au cours du temps pour un objet donné.

instance du composant-description type binaire) ;
cible directe : clientèle expansée (cf. page 53).

néant Il n'est pas possible de terminer cette liste sans citer le cas très particulier de la classe NONE qui est une classe fictive utile pour le graphe de type. Elle hérite automatiquement (et en ignorant tous les conflits) de toutes les classes de l'application (donc, à cause de la non-circularité, aucune classe ne peut en hériter) et n'exporte aucune primitive. De plus, elle ne possède qu'une seule instance, nommée void, qui sert de valeur par défaut à tout objet non initialisé.
– composants-relations valides en tant que :
source directe : non applicable (il n'est pas possible de créer une nouvelle instance du composant-description néant) ;
cible directe : clientèle (cf. page 52).

Enfin, précisons que les classes représentant les tableaux et les chaînes de caractères constituent des cas particuliers en *Eiffel*, mais cela n'est vrai que du point de vue syntaxique : des facilités sont offertes au programmeur pour exprimer des constantes de ces types. Du point de vue sémantique, ces classes sont standards et ne sont pas traitées de manière particulière par les compilateurs, ce qui aurait justifié pour nous la création de deux nouveaux composants-relations.

3.3.3 Les composants-descriptions de *Java*

Nous présentons maintenant dans cette partie treize composants-descriptions pour *Java* comme nous l'avons fait dans la section précédente pour *Eiffel*.

classe C'est un composant-description proche de celui d'*Eiffel*. Prenons comme premier exemple une classe C. Nous pouvons décrire le composant-description correspondant de la manière suivante :
– simple,
– générateur,
– non destructeur (présence d'un ramasse-miettes),
– encapsulant,
– attributs et méthodes autorisés,
– autorise les attributs d'instance ou de description,
– surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte par cette surcharge,
– local au paquetage[23] (C peut être accessible, si elle est public, depuis un autre paquetage grâce à la spécification du nom de son paquetage ou d'une clause d'importation non ambiguë),
– composants-relations valides en tant que :
source directe : héritage inter-classe (cf. page 55) et aussi concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe (cf. page 60) ;
cible directe : héritage inter-classe, agrégation et agrégation de classe.
Nos critères nous permettent d'identifier des différences entre les classes *Eiffel* (cf. section 3.3.2 page 61) et les classes *Java*. Par exemple, nous pouvons remarquer que les classes *Eiffel* gère les attributs d'instance unique

[23] Un paquetage est un ensemble de descriptions réunies par un thème commun.

classe abstraite Il s'agit de classe qui peuvent contenir en même temps des méthodes abstraites et des méthodes concrètes. Une classe abstraite *Java* (mot-clé *abstract*) peut se décrire ainsi :
- simple,
- non générateur (mais peut posséder des constructeurs activables indirectement par un super(...) dans une classe héritière directe concrète),
- non destructeur (présence d'un ramasse-miettes),
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local au paquetage (cf. la remarque correspondante pour les classes *Java* section 3.3.3 page précédente),
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe (cf. page 60) ;

  cible directe : héritage inter-classe, concrétisation (cf. page 57), agrégation et agrégation de classe.

  Là encore nous retrouvons une composant-description proche de celui d'*Eiffel* avec quelques variantes cependant. Le fait est que sur les notions essentielles que sont les classes, l'héritage et l'agrégation, les langages à objets à classes ont beaucoup de choses en commun et quelques divergences. C'est d'ailleurs cela qui nous a permis de mettre en évidence nos critères de comparaison.

interface Les interfaces de *Java* offrent un moyen simple de spécifier un comportement pour un objet sans être, à l'inverse des classes, des conteneurs : une interface ne possède pas d'instance propre. Le composant-description interface est :
- simple,
- non générateur,
- non destructeur,
- encapsulant,
- attributs interdits mais méthodes (abstraites seulement) autorisées,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local au paquetage,
- composants-relations valides en tant que :

  source directe : héritage inter-interface (cf. page 56) et aussi agrégation (cf. page 58), agrégation de classe (cf. page 59), composition[24] (cf. page 59) et composition de classe[25] (cf. page 60) ;

  cible directe : héritage inter-interface, implémentation (cf. page 57), agrégation et agrégation de classe.

[24] Agrégation et composition sont présentes ici en raison des paramètres de méthode et résultats de fonction.

[25] Les interfaces peuvent déclarer des constantes de classes, d'où la présence ici de l'agrégation de classe et de la composition de classe.

classe membre statique Les classes membres statiques ont pour principal intérêt d'apparaître dans l'espace de nommage d'une autre description (cf. section 2.2 page 23). Voyons donc maintenant comment nous pouvons décrire une classe membre statique C. Notons que, de par son statut, C peut accéder aux attributs et méthodes de classe (static) de sa classe encapsulante.
- simple,
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la description encapsulante (mais C peut aussi être accessible, si elle n'est pas private, depuis une autre description grâce à la spécification du nom de sa description encapsulante),
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe (cf. page 60) ;

  cible directe : héritage inter-classe, agrégation et agrégation de classe.

classe membre statique abstraite Les classes membres statiques peuvent elles-aussi être abstraites. Une classe membre statique abstraite est donc, en quelque sorte, une composition d'une classe abstraite (cf. section 3.3.3 page précédente) et d'une classe membre statique (cf. section 3.3.3). Pour les caractéristiques qui diffèrent dans ces composants-descriptions, voici celles que nous pouvons retenir.
- non générateur,
- local à la description encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe (cf. page 60) ;

  cible directe : héritage inter-classe, concrétisation (cf. page 57), agrégation et agrégation de classe.

classe membre Il est possible de définir en *Java* des classes membres qui sont des sortes de sous-description d'une autre description (cf. section 2.2 page 23). Il est à remarquer que chaque instance de C est automatiquement associée à une instance de sa classe encapsulante et a donc accès à tous les attributs et méthodes de celle-ci. Une classe membre C peut être décrite comme suit.
- simple,
- générateur,
- non destructeur,
- encapsulant,

- attributs et méthodes autorisés,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la description encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe[26] (cf. page 60) ;

  cible directe : héritage inter-classe, agrégation et agrégation de classe.

classe membre abstraite Ce sont en même temps des classes abstraites (cf. section 3.3.3 page 66) et des classes membres (cf. section 3.3.3 page précédente). Voici la valeur adéquate pour les caractéristiques qui diffèrent dans ces deux composants-descriptions :

- non générateur,
- local à la description encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe[27] (cf. page 60) ;

  cible directe : héritage inter-classe, concrétisation (cf. page 57), agrégation et agrégation de classe.

classe locale Les classes locales sont internes à une méthode (ou à un initialiseur statique ou non) d'une classe : elles sont ainsi similaires à des variables locales (cf. section 2.2 page 23). Nous pouvons présenter ci-dessous les caractéristiques d'une classe locale C.

- simple,
- générateur,
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- n'autorise que les attributs d'instance[28],
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la méthode encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe[29] (cf. page 60) ;

  cible directe : héritage inter-classe, agrégation et agrégation de classe[30].

[26] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.
[27] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.
[28] *Attribut d'instance* est un abus de langage car il s'agit en fait de variable locale et non d'attribut.
[29] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.
[30] L'agrégation de classe est possible par exemple dans le cas d'une classe locale qui définit une constante de classe du type d'une autre classe locale.

classe locale abstraite Les classes locales abstraites sont évidemment abstraites (cf. section 3.3.3 page 66) et locales (cf. section 3.3.3 page ci-contre). Les points divergents de ces deux composants-descriptions trouvent ici les caractéristiques suivantes.
- non générateur,
- n'autorise que les attributs d'instance,
- local à la méthode encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe[31] (cf. page 60) ;

  cible directe : héritage inter-classe, concrétisation (cf. page 57), agrégation et agrégation de classe[32].

classe anonyme Les classes internes anonymes sont comme des classes locales auxquelles le nom a été retiré[33]. Elles sont donc utilisées directement dans des expressions (cf. section 2.2 page 23). Une classe anonyme C peut être décrite par les capacités suivantes.
- simple,
- générateur (une seule instance est générée pour C et elle l'est automatiquement),
- non destructeur,
- encapsulant,
- attributs et méthodes autorisés,
- attribut d'instance unique (l'instance de C est anonyme et créée au sein d'une expression, elle n'est donc pas partageable),
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à l'objet issu de l'expression encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-classe (cf. page 55) et aussi concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe[34] (cf. page 60) ;

  cible directe : agrégation et agrégation de classe[35].

interface membre statique Elles sont l'équivalent, du point de vue des interfaces, des classes membres statiques (cf. section 2.2 page 23). Notez que les interfaces membres statiques sont les seules interfaces servant de sous-description

[31] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[32] L'agrégation de classe est possible par exemple dans le cas d'une classe locale qui définit une constante de classe du type d'une autre classe locale.

[33] On peut ainsi les rapprocher des lambda-expressions de *Lisp* et elles sont très utiles quand il s'agit, par exemple, d'implémenter une interface pour n'exécuter qu'en un seul endroit le code en question.

[34] L'agrégation de classe et la composition de classe sont permises pour la définition des constantes.

[35] L'agrégation de classe est possible en programmant une variable de classe initialisée par un objet issu d'une classe anonyme.

en *Java* : il n'y a pas d'interface membre (non statique), d'interface locale ou d'interface anonyme. Imaginons donc une interface membre statique C et voyons comment nous pouvons la décrire.
- simple,
- non générateur,
- non destructeur,
- encapsulant,
- attributs interdits mais méthodes (abstraites) autorisées,
- autorise les attributs d'instance ou de description,
- surcharge autorisée, le type de la valeur de retour n'étant pas pris en compte,
- local à la description encapsulante,
- composants-relations valides en tant que :

  source directe : héritage inter-interface (cf. page 56) et aussi agrégation (cf. page 58), agrégation de classe (cf. page 59), composition[36] (cf. page 59) et composition de classe[37] (cf. page 60) ;

  cible directe : héritage inter-interface, implémentation (cf. page 57), agrégation et agrégation de classe.

tableau Il s'agit d'un cas particulier de *Java*. Les tableaux sont manipulables par la classe java.lang.reflect.Array et sont modélisées par des pseudos-classes créées automatiquement lors de la création de leur première instance[38] (cf. section 2.2 page 23). Une description tableau peut être caractérisée comme suit.
- simple (une description est créée pour chaque type d'élément : un tableau d'entiers n'est pas du même type qu'un tableau de booléens[39]),
- générateur,
- non destructeur,
- encapsulant,
- attributs (en fait des pseudo-attributs tels length) et méthodes (en fait des opérateurs) autorisés,
- partageur,
- rien n'est surchargé dans cette pseudo-classe,
- global,
- composants-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer une nouvelle instance du composant-description tableau) ;

  cible directe : agrégation (cf. page 58) et agrégation de classe (cf. page 59).

type primitif Ils constituent un second cas particulier pour *Java* et sont au nombre de huit : booléen, caractère, octet, entier court, entier, entier long, flottant et flottant double. Contrairement à tous les autres, les types primitifs ne sont pas des objets (cf. section 2.2 page 23). Voici leurs caractéristiques.

[36] Agrégation et composition sont présentes ici en raison des paramètres de méthode et résultats de fonction.

[37] Les interfaces peuvent déclarer des constantes de classes, d'où la présence ici de l'agrégation de classe et de la composition de classe.

[38] Les tableaux *Java* sont donc des objets.

[39] On peut donc voir la pseudo-classe tableau de *Java* comme générique, le paramètre de généricité étant le type des éléments.

- Ces types disposent d'une interface syntaxique particulière : des mots-clés leur sont associés et il est possible d'utiliser directement des constantes de leur type (exemples : 16, true, ...).
- Chaque type primitif décrit une valeur et non une référence. Invoquer une primitive pour lui n'a donc pas de sens.
- Chaque type primitif est également décrit par un véritable classe (exemple : Integer pour int) ce qui permet d'utiliser des valeurs primitives en tant qu'objet (descendant d'Object).
- composants-relations valides en tant que :

  source directe : non applicable (il n'est pas possible de créer une nouvelle instance du composant-description type primitif) ;

  cible directe : composition (cf. page 59) et aussi composition de classe (cf. page 60).

Comme pour *Eiffel*, les classes représentant les chaînes (de caractères) ne constituent un cas particulier que du point de vue syntaxique.

3.4 Bilan

Ce chapitre nous a permis d'introduire les notions de composant-relation et de composant-description qui forment, nous le verrons, des entités de base du modèle *OFL*. Il a ensuite consisté à réaliser, sous le point de vue de ces composants, l'étude exhaustive des langages *Eiffel* et *Java*.

De cette étude, nous pouvons tirer les enseignements suivants :

- Nous avons pu décrire de manière satisfaisante les composants-relations et composants-descriptions grâce à de nombreuses questions simples. Cela va nous amener à définir un système de paramètres pour décrire les comportements de tels composants.
- Les composants-relations d'importation ne peuvent pas aisément être décrits de la même manière que les composants-relations d'utilisation. Cette remarque induira une différence entre les paramètres des importations et ceux des utilisations, bien que comme nous le verrons, il soit pertinent de partager bon nombre d'entre eux.
- La diversité des composants-relations (respectivement : composants-descriptions) est bien plus importantes qu'elle ne le paraît *a priori* : nous aurions facilement tendance à simplifier en prétendant qu'il n'existe que la relation d'héritage (respectivement : de classe), ce qui est faux. L'ajout de composants-relations et composants-descriptions à un langage, s'il se fait de façon limitée et contrôlée, relève de la même réflexion.
- L'étude que nous venons de réaliser des langages *Eiffel* et *Java* l'a été pour déterminer un certain nombre de critères de comparaison des sortes de descriptions et de relations entre descriptions. Les résultats de cette étude sont amenés à être généralisés pour couvrir un large panel de langages à objets à classes. Comme nous le verrons dans la suite de ce document, les paramètres que nous définissons permettent d'ailleurs de représenter un éventail de composants-relations et composants-descriptions bien plus important que ceux de ces deux seuls langages.

Le processus de spécialisation des sortes de description et relation engagé par les concepteurs de *Java* (avec les interfaces et la relation d'implémentation) nous semble

prometteur. Il permet en effet de mieux préciser la volonté du programmeur lorsqu'il écrit ses applications. C'est dans cette optique que nous allons maintenant décrire le modèle *OFL*.

Chapitre 4

OFL et ses concepts

4.1 Introduction

Avant de vous décrire de manière détaillée, dans le chapitre 5 page 103, l'architecture du modèle *OFL*, nous allons vous présenter, dans le présent chapitre, les trois principaux concepts de ce modèle, issus de nos études bibliographiques et préliminaires.

Nous vous proposons tout d'abord quelques définitions en accord avec la figure 4.1[1].

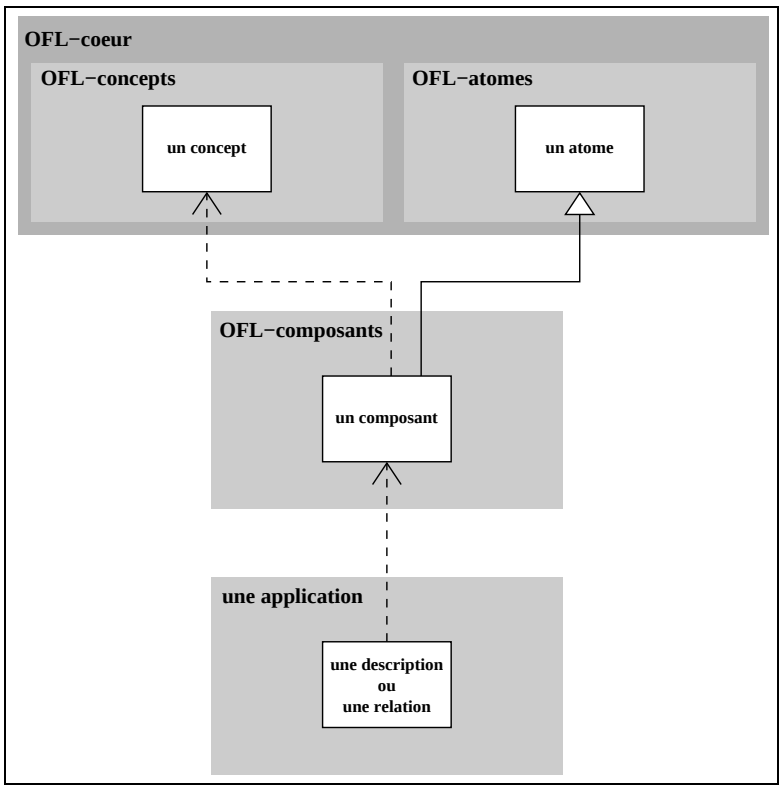

FIG. 4.1 – Architecture simplifiée d'*OFL*

Le modèle OFL peut être défini simplement en trois partie : l'*OFL*-cœur (cf. dé-

[1]Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48.

finition 9), les *OFL*-composants (cf. définition 12) et les applications. Nous verrons dans la section 5.1 page 103 que d'autres partie existent, mais il n'est ni utile ni très facile de les présenter si tôt.

Définition 9 (*OFL*-cœur) *L'OFL-cœur contient les entités de base du modèle, celles qui sont fournies aux méta-programmeurs et programmeurs, indépendamment de tout langage. L'OFL-cœur renferme principalement deux sortes d'entité : les OFL-concepts (cf. définition 10) et les OFL-atomes (cf. définition 11).*

Définition 10 (*OFL*-concept) *Un OFL-concept est un méta-OFL-composant (cf. définition 12).*

Définition 11 (*OFL*-atome) *Un OFL-atome est un OFL-composant (cf. définition 12) général, non spécifique à un langage donné. Un OFL-atome est à la fois un OFL-composant et une entité de l'OFL-cœur.*[2]

Définition 12 (*OFL*-composant) *Un OFL-composant est une méta-entité pour un langage de programmation. On retrouve notamment, au sein des OFL-composants, les composants-langages, les composants-descriptions et les composants-relations mais aussi la réification des méthodes, des expressions, des objets, ... Les OFL-composants sont constitués des OFL-atomes (cf. définition 11) qui sont généraux, et des composants spécifiques à un langage donné.*

4.2 Principes de base

4.2.1 Hyper-généricité

Définition 13 (hyper-généricité) *L'hyper-généricité est le paramétrage du comportement d'un système informatique selon des critères simples.* Le terme *hyper-généricité* est utilisé par extension du terme *généricité* qui, dans les langages à objets, décrit la capacité d'une classe à être paramétrique.

Nous venons de décrire rapidement les principales entités de base du modèle *OFL* (les concepts, les atomes et les composants) puis de présenter une étude d'*Eiffel* et de *Java*. Nous avons enfin fait part de notre volonté de généraliser notre démarche à l'ensemble ou tout au moins à un large ensemble de langages à objets à classes. Nous avons également ébauché une première présentation des trois concepts principaux du modèle (concept-description, concept-relation et concept-langage) et de leurs instances (respectivement composants-descriptions, composants-relations et composants-langages).

Notre volonté, tout au long de la genèse du modèle *OFL*, a été de permettre au méta-programmeur de modifier le comportement, la sémantique opérationnelle, des composants de son langage. Cependant, réaliser tout cela par méta-programmation serait une tâche phénoménale. Nous avons donc choisi de mettre en œuvre le principe d'hyper-généricité [50] (cf. définition 13). Nous avons sélectionné un ensemble de paramètres dont la valuation permet de décrire le comportement d'un composant. Un système d'actions (cf. définition 14 page ci-contre) prédéfini exploite la valeur de ces paramètres.

[2] Sur la figure 4.1 page précédente, les *OFL*-atomes sont exclus du cadre *OFL-composants* dans le seul but de ne pas alourdir le schéma.

Définition 14 (action) *Une action, dans le modèle OFL, est une routine qui met en œuvre un comportement des langages décrits par le modèle. L'algorithme d'une action tient compte de la valeur des paramètres hyper-génériques (cf. définition 13 page précédente).* La recherche de la méthode adéquate lors par la liaison dynamique, l'envoi de message, ou encore l'affectation d'un objet à un attribut sont des exemples d'action.

Chaque instance de concept possède donc des paramètres. Seuls les composants-relations et composants-descriptions sont également équipés d'actions (les actions sont partagées par tous les composants-relations ou composants-descriptions, ces derniers pouvant les redéfinir localement pour les affiner) et d'un protocole pour les méta-programmer. Le travail du méta-programmeur peut prendre deux formes : soit il attribue simplement une valeur aux paramètres décrits dans les concepts et définit ainsi les composants de son langage, soit il souhaite modifier la sémantique associée à au moins un de ces paramètres. Dans le premier cas, son travail est terminé et c'est là l'un des intérêts de notre approche. Dans le second cas, qui ne doit s'appliquer qu'en dernier ressort, il lui faut redéfinir les actions dont l'algorithme ne convient pas.

Ces actions ont été classées en six thèmes :

1. la recherche de primitive (cf. section 6.4 page 120, exemple : recherche d'une primitive dans le cadre de la liaison dynamique),
2. l'exécution de primitive (cf. section 6.5 page 125) ,
3. le contrôle (cf. section 6.6 page 130, exemple : vérification des paramètres effectifs d'un appel de primitive par rapport aux paramètres formels),
4. la gestion d'instance de description (cf. section 6.7 page 138, exemple : création d'une instance),
5. la gestion d'extension de description (cf. section 6.8 page 141) et
6. les opérations de base (cf. section 6.9 page 143, exemples : l'affectation ou la copie d'objet).

Les actions de concept-description font un usage intensif de celles de concept-relation. Il n'est donc pas étonnant de constater que les actions de concept-description sont distribués en six thèmes reprenant ceux des concepts-relations. De manière générale, le même nom d'action est utilisé pour les concepts-descriptions et concepts-relations, celle se trouvant dans les premiers appelant souvent celles qui se trouvent dans les seconds, le tout pour réaliser une opération unique (telle la liaison dynamique par exemple).

4.2.2 Structure des *OFL*-concepts

Données Un concept est un méta-composant ; à ce titre un concept décrit la structure (données) et le comportement (paramètres et actions, cf. ci-dessous) d'un composant. Dans les sections suivantes, les sections « Données » ne font que préfigurer la présentation plus complète que nous verrons dans la section 5.2 page 108 sur les atomes.

Paramètres Les concepts-relations[3] définissent des paramètres. De plus, pour chaque composant-relation, chaque paramètre général (au sens où il est toujours présent quel que soit l'usage qui est fait du composant-relation) possède une

[3] À moins que nous précisions explicitement le contraire, lorsque nous faisons référence dans la suite de cette section à un concept-relation, il est toujours possible d'ajouter « (respectivement concept-description) ». Les concepts-langages définissent aussi des paramètres, mais ceux-ci sont peu importants.

valeur qui décrit un aspect de la sémantique opérationnelle de ses relations. Chaque composant-relation peut, de plus, voir sa sémantique complétée par des paramètres plus spécifiques (décrits dans le concept-relation) pour des applications mettant en œuvre un service spécial comme la persistance ou la concurrence. Pour chaque paramètre, nous expliquons son sens, les valeurs qu'il peut prendre et dans quelle mesure nous pouvons le redéfinir, dans l'idée que nous pouvons réduire une propriété mais jamais l'étendre.

Les paramètres forment un ensemble d'informations qui sera mis à profit par le système d'actions.

Redéfinitions Chaque instance de concept peut aussi redéfinir les valeurs des paramètres des instances de concepts de plus bas niveau qu'il utilise : nous estimons que les composants-langages constituent le niveau le plus élevé de la modélisation, viennent ensuite les composants-descriptions et enfin les composants-relations. Ainsi, un composant-langage peut redéfinir la valeur des paramètres des composants-descriptions et composants-relations qu'il utilise et un composant-description peut redéfinir celle des paramètres des composants-relations qu'il utilise. Les redéfinitions permettent ainsi de réutiliser un composant prévu pour réaliser des tâches complexes dans un but plus simple sur le principe de la limitation, du « qui peut le plus peut le moins ». Nous pouvons, par exemple, imaginer un composant-description qui réutilise un composant-relation d'héritage multiple en occultant sa capacité *multiple*.

Cette idée de limitation est en fait dépendante de la sémantique de chaque paramètre. En première approximation, nous pourrions considérer que, pour tous les types disposant d'un ordre (exemple : integer), limiter signifie sélectionner une valeur inférieure ou égale. Mais cela nous amènerait à considérer par exemple que les paramètres booléens ne peuvent être redéfini que vers false, ce qui peut être un contresens. Imaginons en effet un paramètre Simple qui définit si un composant-relation est simple ou multiple. Il est évident que pour lui la limitation consiste à passer de false à true et non le contraire. Nous avons donc préféré, pour chaque paramètre, spécifier dans quelle mesure il peut être redéfini, cela nous permettant de conserver toute latitude pour associer une sémantique aux valeurs données.

Imaginons par exemple, pour illustrer les redéfinitions, un composant-relation CR possédant un paramètre Circularity (cf. section 4.4.2 page 87) avec la valeur allowed, ce qui signifie qu'il est permis de réaliser un cycle avec des relations instances de ce composant-relation. Prenons maintenant un composant-langage CL qui utilise CR. Il est permis grâce au dispositif indiqué ci-dessus de spécifier dans CL le triplet <CR, Circularity, forbidden> pour interdire, dans le langage décrit, les cycles. Mais, comme nous l'avons déjà indiqué, nous ne pouvons, dans ce cas, permettre de passer de forbidden à allowed, car cela est interdit dans la définition de ce paramètre (cf. section 4.4.2 page 87).

Assertions Pour chaque composant-relation, nous pouvons établir la liste de contrôles qui doivent être vérifiés dans les descriptions-sources et descriptions-cibles pour que la relation puisse être posée. Nous utilisons aussi les assertions pour vérifier, dans chaque composant-description, quels sont les composants-relations compatibles et sous quelles conditions. De même, il nous est possible de poser et tester des contraintes d'association entre des composants-descriptions et des composants-relations au sein des composants-langages. Pour les

composants-relations (respectivement : les composants-descriptions et les composants-langages), nous disposons de deux sortes d'assertions :

1. **Les contraintes d'utilisation de ce concept-relation (ou concept-description ou concept-langage)** que le méta-programmeur peut modifier ou compléter, et
2. **Les contraintes structurelles applicables à tout concept-relation (respectivement : concept-description et concept-langage)**. Ces dernières constituent donc le modèle *OFL* lui-même et ne sont pas sujettes à modification.

Des assertions, formules logiques annotant le modèle et son code, sont donc associées à la notion de concept-langage, comme d'ailleurs à celles de concept-description et concept-relation. Elles permettent de préciser la sémantique de chaque composant logiciel et de contrôler, au cours de son utilisation, que l'usage qui en est fait est conforme à la spécification donnée[4]. Nous retrouvons dans *OFL* les trois principales sortes d'assertions : invariants (condition qui doit être vérifiée à tout moment), préconditions (condition qui doit être vérifiée avant l'exécution d'une méthode) et postconditions (condition qui doit être vérifiée après l'exécution d'une méthode) [76].

Actions Les paramètres décrivent la sémantique sous forme de valeurs (exemple : le paramètre Circularity peut prendre les valeurs allowed ou forbidden). Pour décrire une sémantique opérationnelle, par définition, des valeurs sont insuffisantes. Pour chaque concept-relation, nous proposons donc, en plus des paramètres, un système d'actions. Il s'agit d'algorithmes[5] qui décrivent des comportements clés (exemples : la recherche ou l'exécution de primitive) en fonction des valeurs des paramètres. Avec l'idée d'actions implantées plus tard par des méthodes, nous pouvons imaginer leur corps comme une succession d'alternatives prenant en compte les valeurs possibles des paramètres (seuls ceux qui sont pertinents pour l'opération à réaliser, bien entendu) pour effectuer le travail souhaité. Chaque composant-relation peut, comme pour ses paramètres, voir sa sémantique complétée par des actions plus spécifiques, ce cas ne devant se présenter que dans de très rares occasions.

Il nous faut également préciser une particularité des actions de concept-relation. Il est souvent nécessaire d'exécuter ces opérations en plusieurs étapes. Aussi ces actions peuvent-elles être découpées en sous-actions qui interviennent à différents moments de l'exécution (cf. section 6.2 page 118).

La description complète des actions étant longue, nous la donnons dans la section 6 page 117 indépendamment de la présentation des concepts-descriptions et concepts-relations.

4.3 Concept-description

Les instances de concepts-descriptions (nommées, nous l'avons déjà vu, composants-descriptions) représentent, ainsi que nous l'avons expliqué précédemment, la

[4]Ces assertions, décrites au niveau des composants d'un langage de programmation, peuvent donc être considérées comme des méta-assertions pour un programmeur utilisant ce langage.

[5]Pour éclaircir les choses, bien que cela soit un choix d'implémentation du modèle *OFL* qui n'a pas lieu d'être ici, nous pouvons aisément nous figurer que les actions sont des routines, autrement dit des méthodes si *OFL* est programmé dans un langage à objets.

notion de classe (et de tout ce qui y ressemble) dans les langages à objets. Bien que notre but premier ne soit pas de paramétrer la sémantique opérationnelle des descriptions (car nous souhaitons nous concentrer sur les relations entre descriptions), nous avons défini un ensemble de paramètres décrits dans la présente section. Leur définition est en effet fort utile à la mise en place d'un système cohérent de paramétrage des relations qui doivent les lier. Le paramétrage des descriptions participe aussi évidemment à une meilleure définition de la sémantique opérationnelle des langages.

4.3.1 Données

Dans cette section, nous introduisons des éléments de réification qui seront plus largement présentés dans la section 5.2 page 108. Ces éléments sont utiles à la compréhension des paramètres spécifiés dans la section 4.3.2 page ci-contre.

Les informations de réification peuvent être classés en deux partie :

1. **Celles qui sont associées à un composant-description** :
   - Chaque composant-description est tout d'abord décrit par les ensembles de composants-relations qui peuvent lui être attachés :
     (a) la liste des composants-relations valides pour prendre le composant-description comme source.
     (b) la liste des composants-relations valides pour prendre le composant-description comme cible.

     Tout comme nous le verrons pour les concepts-relations, ces ensembles ne prennent tout leur sens que dans le contexte de la définition d'un langage de programmation bien déterminé. En effet, si nous nous trouvons dans la situation de décrire un composant-description de bibliothèque (et donc réutilisable), nous ne pouvons *a priori* connaître tous les composants-relations disponibles.
   - Chaque composant-description référence également le langage auquel il appartient, si cela est signalé par le paramètre Context décrit dans la section suivante.
   - Chaque composant-description définit une liste de qualifieurs valides pour ses instances (qui sont des descriptions), pour les attributs et méthodes (dans les descriptions), pour les paramètres (de méthode) et pour les assertions.
2. **Celles qui sont associées à une description (instance de composant-description)** :
   - À chaque description est associé un ensemble de primitives. Ce sont tout simplement les primitives décrites dans cette description : ses attributs et méthodes.
   - Nous trouvons également pour chaque description un ensemble de relations pour lesquelles elle est utilisée comme source et un ensemble de relations pour lesquelles elle sert de cible.
   - Nous retrouvons aussi, pour les descriptions génériques, la liste des types contraignant les paramètres génériques effectifs. Il est ainsi possible de gérer des classes génériques telles celles d'*Eiffel* (cf. section 3.3.2 page 63). Chaque description peut être aussi associée à un ensemble de types. Souvent cet ensemble est réduit à un seul type : nous disons qu'une classe représente un type, sauf si elle est générique et en représente alors plusieurs.

Enfin, ajoutons que chaque concept-description possède la connaissance de son extension (un ensemble de composants-descriptions), et qu'il en est de même pour chaque composant-description (dont l'extension est un ensemble de descriptions).

4.3.2 Paramètres

Voici la liste de paramètres qui nous permettent de définir la sémantique d'une instance de concept-description :

Name Il s'agit du nom du composant-description. Il doit être unique au sein d'un langage et ne peut pas être redéfini.

Context Les composants-descriptions peuvent être décrits dans deux contextes différents :

1. un monde fermé où le méta-programmeur a connaissance de tous les composants-descriptions et composants-relations mis en œuvre (c'est le cas lors de la modélisation d'un langage) ou
2. un monde ouvert où les composants-descriptions et composants-relations ne sont pas *a priori* connus (par exemple, lors de l'implantation d'une bibliothèque de composants-descriptions).

Ce paramètre indique dans laquelle de ces deux situations se trouve la personne qui décrit cette entité : language ou library.

Ce paramètre peut être librement redéfini, mais il est alors à la charge du méta-programmeur de vérifier la cohérence du composant-description, notamment en contrôlant la validité des listes de composants-relations valides que nous avons vu dans la section 4.3.1 page ci-contre. Pour réaliser cette vérification, le méta-programmeur peut faire appel à une assertion telle celles de la section 4.3.4 page 81.

Persistence Un booléen signale l'utilisation (true) ou non (false) de la classe de services de la persistance.

Si cette classe de service est activée, des paramètres (certains paramètres de concept-relation spécifiques à la persistance sont décrits section 4.4.2 page 91) et actions spécifiques entrent alors en jeu.

Ce paramètre Persistence peut être redéfini de true vers false.

Concurrency Autre classe de services, autre paramètre ; celui-ci indique évidemment s'il doit être tenu compte de la concurrence.

Il est également booléen, s'utilise de manière similaire à Persistence et peut être redéfini de true vers false.

... Pour chaque classe de services intégrée à *OFL*, telle la persistance ou la concurrence que nous venons de voir, un paramètre booléen signale l'éventuelle activation.

Les redéfinitions sont également permises selon le même principe : de true vers false. L'implantation de cette liste de classes de service pourra être faite sous la forme d'un paramètre regroupant un ensemble de classes de service plutôt que d'un paramètre par classe de service.

Genericity Ce paramètre booléen spécifie si la description est générique (true), c'est-à-dire si elle possède la capacité de représenter plusieurs types, ou pas (false).

Il est possible de le redéfinir de true vers false.

Generator Ce paramètre indique si le composant-description est générateur, c'est-à-dire s'il peut créer des instances.

Dans ce cas, il sera par exemple dans la mesure de fournir des constructeurs (allocateurs) et/ou des initialiseurs. Ce paramètre prend la forme d'un booléen, true signifiant générateur, false non.

Il peut être redéfini de true vers false.

Destructor C'est le pendant du précédent, il indique s'il est possible de désallouer une instance.

Il peut être redéfini de true vers false.

Extension_creation Ce paramètre permet de déterminer si, lors de la création de la description, son extension est calculée automatiquement (automatically) ou manuellement (manually).

Toutes les descriptions d'*Eiffel* et de *Java* présentées dans les sections 3.3.2 page 61 et 3.3.3 page 65 peuvent être considérées comme possédant des extensions créées automatiquement vides. Par la suite, nous considérons généralement dans ces langages que l'extension d'une description est l'union des extensions de ses descriptions héritières.

Ce paramètre peut être redéfini librement.

Encapsulation Il s'agit ici de préciser si les primitives sont encapsulées dans les descriptions[6].

Ce paramètre est en fait une paire de deux booléens précisant si l'encapsulation est effective (true pour oui, false pour non) d'abord pour les attributs, puis pour les méthodes.

Ce paramètre peut être redéfini librement.

Sharing_control Ce paramètre indique si les instances de cette description peuvent être partagées et si oui, de quelle manière.

Il s'agit d'une liste de valeurs exprimant les possibilités de partage parmi :
- description : L'instance peut être partagée par toutes les instances d'une description (notion de *variable de classe*).
- instance : L'instance peut être partagée par plusieurs instances (notion de *variable d'instance*).
- unique_instance : L'instance ne peut pas être partagée, elle est donc spécifique à une seule autre instance (notion de *variable d'une seule instance*).

Ce paramètre peut être redéfini en retirant un ou plusieurs éléments à la liste de valeurs. Remarquons que cette liste doit toujours contenir au moins une valeur pour conserver un sens.

Visibility Nous indiquons ici depuis quelles entités la description est visible.

Les valeurs possibles de ce paramètre sont :
- global : Ici, la description est visible de tout point de l'application.
- package : La visibilité est limitée au paquetage (ensemble de descriptions) englobant.
- description : La visibilité est limitée à la description encapsulante. Nous avons donc ici affaire à une sous-description.
- method : La visibilité est limitée à la méthode englobante.
- statement : La visibilité de cette description est ici très faible puisqu'elle se limite à l'instruction englobante.

[6] La gestion de méthodes non encapsulées est utile pour décrire des langages tels *CLOS*.

– expression : La visibilité est encore plus limitée, à l'expression englobante.
– list : Cette description n'est accessible qu'aux éléments d'une liste généralement spécifiée par le concepteur de la description.

Ce paramètre peut être redéfini librement.

Attribute Ce paramètre permet de déterminer si la description admet des attributs (allowed) ou non (forbidden).

Dans ce dernier cas, cela signifie que seules des méthodes peuvent être incluses.

Il est possible de redéfinir ce paramètre de allowed vers forbidden.

Method Ce paramètre permet de déterminer si la description admet des méthodes (allowed) ou non (forbidden).

Il est possible de redéfinir ce paramètre de allowed vers forbidden.

Overloading Ce paramètre indique si le composant-description admet (allowed) ou interdit (forbidden) la surcharge des primitives. Ce paramètre est une composition de quatre valeurs qui indique la possibilité de surcharge pour les attributs, les résultats de fonction, le nombre des paramètres de méthode et le type des paramètres de méthode. Ces valeurs se combinent pour donner des solutions traditionnelles telles celles d'*Eiffel* (<forbidden, forbidden, forbidden, forbidden>) et de *Java* (<forbidden, forbidden, allowed, allowed>), ou encore toutes celles que voudrait inventer le méta-programmeur.

Chaque membre de ce quadruplet de valeurs peut être redéfini de allowed vers forbidden.

4.3.3 Redéfinitions

Comme nous l'avons vu dans la section 4.2.2 page 75, il est possible dans un composant-description de redéfinir des paramètres de composants-relations dans la mesure ou la redéfinition n'entraine qu'une diminution des capacités de ces derniers.

Imaginons un composant-relation nommé Inheritance et décrivant un héritage multiple avec gestion de la persistance. La valeur de son paramètre Cardinality est donc $1 - \infty$ et Persistence est égal à true. Prenons maintenant un composant-description Class standard qui souhaite utiliser un héritage simple ne gérant pas les problèmes de persistance. Il lui suffit donc simplement de redéfinir, pour le composant-relation Inheritance, les paramètres Cardinality à $1 - 1$ et Persistence à false. Remarquez que ces deux redéfinitions sont réalisées conformément aux possibilités décrites dans les deux paramètres de composant-relation (cf. section 4.4.2 page 85).

Un composant-description a également la possibilité de redéfinir à la baisse les qualifieurs présents dans une composant-relation. Ces qualifieurs s'appliquent à une relation.

4.3.4 Assertions

Les composants-descriptions sont contrôlés par des invariants. Voici deux exemples :

1. **Une contrainte structurelle applicable à tout composant-description.** Nous souhaitons vérifier que, pour toute relation liant une description issue de ce composant-description, si le composant-relation dont est issue cette relation est multiple alors il n'est jamais utilisé pour réaliser plusieurs relations simples parallèles. Disons que si nous disposons, par exemple, d'un héritage multiple, le

composant-description en question exige que nous fassions usage d'une seule relation multiple plutôt que d'une composition de relations simples. Cela pourrait s'exprimer ainsi en français. « Soient D une description et CR un composant-relation pour lequel Cardinality est égal à $1 - \infty$, alors une instance de D ne peut pas être source directe de plus d'une instance de CR. » Les relations indirectes ne posent pas problème ici.

2. **Une contrainte d'utilisation d'un concept-description spécifique.** Imaginons maintenant un concept-description de bibliothèque (cf. section 4.3.2 page 79) qui n'est donc pas associé précisément à un langage donné mais plutôt destiné à servir de composant de base pour un méta-programmeur. Ce concept-description décrit une classe concrète finale, c'est-à-dire qu'elle ne doit servir de cible à aucune relation d'importation. Nous pouvons alors aisément spécifier un invariant pour les description issue de ce concept-description particulier. « Soient D une description et CR un composant-relation pour lequel Kind (cf. section 4.4.2 page 85) est égal à import, alors aucune instance de D ne peut être la cible d'une instance de CR. » Nous pourrions, au premier abord, penser qu'il aurait suffit de ne pas spécifier, dans la liste des relations acceptées en tant que cible directe, les composants-relations d'importation. Mais cela n'est pas applicable pour un composant de bibliothèque qui n'a pas, par avance, connaissance des composants-relations qui existent pas ailleurs.

4.3.5 Bilan sur les concepts-descriptions

Les tableaux 4.1 page suivante et 4.2 page 84 proposent une récapitulation de la structure d'un concept-description.

4.4 Concept-relation

La seconde des trois entités qui constituent le cœur d'*OFL* est, comme nous l'avons déjà vu, le concept-relation.

4.4.1 Données

Nous rappelons que, dans cette section, nous introduisons des éléments de réification qui seront plus largement présentés dans la section 5.2 page 108.

Les informations de réification peuvent être classés en deux partie :

1. **Celles qui sont associées à un composant-relation** :
   - Chaque composant-relation, instance d'un concept-relation, dispose de deux ensembles de composants-descriptions :
     (a) les composants-descriptions valides en tant que source directe, c'est-à-dire ceux que la relation accepte en tant que source.
     (b) les composants-descriptions valides en tant que cible directe.

     Rappelons que dans un contexte de composants de bibliothèque, où l'intégralité des composants-descriptions disponibles n'est pas forcément connue, ces deux listes ne sont qu'indicatives et peuvent donc être complétées voire redéfinies.

[7] Les propriétés spécifiques à chaque description sont développées et complétées dans la section 5.2.8 page 114.

Informations générales au modèle *OFL*		
Assertion	**Type**	**Exemple de valeur**
Structural invariants	set of condition	cf. exemple page 81
Action	**Type**	**Exemple de valeur**
Actions	set of routine	*cf. page 117*

Informations spécifiques au concept-description		
Élément de réification	**Type**	**Exemple de valeur**
Extension	set of description-component	{Class, Interface}

Informations spécifiques à chaque composant-description		
Élément de réification	**Type**	**Exemple de valeur**
Language	language	Java-like
Valid source relationship-components	set of relationship-component	{Inheritance, Aggregation}
Valid target relationship-components	set of relationship-component	{Inheritance, Aggregation, Implementation}
Valid Qualifiers	set of <atom_name, set of string>	{<description, {}>, <attribute, {"static"}>}
Extension	set of description	{Class1, Class2, Interface1}
Paramètre	**Type**	**Exemple de valeur**
Name	string	"Java-class"
Context	*language* | *library*	library
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Genericity	boolean	false
Generator	boolean	true
Destructor	boolean	false
Extension_creation	*automatically* | *manually*	automatically
Encapsulation	<boolean, boolean>	<true, true>
Sharing_control	set of (*description* | *instance* | *unique_instance*)	{*description*, *instance*}
Visibility	*global* | *package* | *description* | *method* | *object* | *statement* | *expression* | *list*	global
Attribute	*allowed* | *forbidden*	allowed
Method	*allowed* | *forbidden*	allowed
Overloading	<*allowed* | *forbidden*, *allowed* | *forbidden*, *allowed* | *forbidden*, *allowed* | *forbidden*>	<forbidden, forbidden, allowed, allowed>
à suivre dans le tableau 4.2 page suivante		

TAB. 4.1 – Récapitulatif de la structure de concept-description (1/2)

suite et fin du tableau 4.1 page précédente		
Informations spécifiques à chaque composant-description		
Redéfinition	**Type**	**Exemple de valeur**
Relationship limitations	set of <relationship-component, parameter-name, value>	{<cr1, Circularity, forbidden>, <cr2, Polymorphism_implication, none>}
Valid-Qualifiers redefinitions	set of <relationship-component, atom_name, set of string>	{<cr1, relationship, {}>}
à suivre dans le tableau 4.2		
Assertion	**Type**	**Exemple de valeur**
Local invariants	set of condition	cf. exemples page 81

Informations spécifiques à chaque description[7] (cf. section 5.2.8 page 114)		
Élément de réification	**Type**	**Exemple de valeur**
Source relationships	set of relationship	{Inheritance18, Inheritance19, Aggregation3}
Target relationships	set of relationship	{Aggregation4, Aggregation5, Aggregation6}
Features	set of feature	{Prim1, Prim2}
Formal generic types	set of type	{object}
Effective types	set of type	{ListOfObjects, ListOfCircles, ListOfMen}

TAB. 4.2 – Récapitulatif de la structure de concept-description (2/2)

- Chaque composant-relation définit une liste de qualifieurs possibles pour leurs instances (qui sont des relations).

2. **Celles qui sont associées à une relation (instance de composant-relation)** :
   - Chaque relation a connaissance des ensembles de descriptions (en fait des types qui sont définis par ses descriptions) qui lui servent de sources et de cibles. Et comme les composants-descriptions, les composants-relations conservent le langage auquel ils appartiennent si cela est indiqué par le paramètre Context décrit dans la section suivante.
   - Chaque relation décrit également la manière dont elle gère les primitives de ses descriptions-cibles. Il est ainsi possible d'avoir connaissance des primitives supprimées, masquées, démasquées ou rendue abstraites ou concrètes par la relation. Pour les primitives renommées, l'ancienne signature et le nouveau nom sont conservés. Pour celles qui sont redéfinies l'ancienne et la nouvelle primitive sont disponibles.

De plus, à l'image des concepts-descriptions, les concepts-relations et les composants-relations connaissent leur extension.

4.4.2 Paramètres

Paramètres généraux

Les paramètres de concept-relation fournissent l'information essentielle pour permettre une exécution des actions conforme aux souhaits du méta-programmeur. En voici la liste :

Name C'est simplement le nom du composant-relation. Il doit être unique au sein d'un langage et ne peut pas être redéfini.

Kind Ce paramètre est important, il définit le genre principal de la relation :
- importation inter-description (import) ou
- utilisation inter-description (use) ou
- entre type(s) et objet(s) (type-object) ou
- entre objets (objects).

Ce paramètre ne peut pas être redéfini.

Context Il signale si le composant-relation est défini dans le cadre d'un langage spécifique (language) et donc dans un monde fermé ou comme un composant de bibliothèque (library).

Dans ce dernier cas, les listes de composants-descriptions compatibles données ne doivent être considérées que comme des indications et peuvent être complétées voire redéfinies. Les valeurs de plusieurs paramètres doivent tenir compte de cette distinction. Par exemple, Polymorphism_policy est indifférent à cela alors que Opposite ne l'est pas : si le monde est ouvert, nous ne pouvons pas savoir *a priori* si un composant-relation prochainement ajouté ne fournira pas un inverse à un composant-relation existant.

Ce paramètre peut éventuellement être redéfini librement mais avec prudence, l'adéquation du composant-relation à son nouveau contexte devant être assurée par le méta-programmeur.

Persistence Ce paramètre booléen indique si le composant-relation doit intégrer et tenir compte des paramètres et actions concernant la classe de services persistance.

Il peut être redéfini de true vers false.

Concurrency Il indique, comme le précédent et comme pour les composants-descriptions, l'activation d'une classe de services, ici la concurrence.

Il peut être redéfini de true vers false.

... Pour chaque classe de services, nous avons donc un paramètre booléen qui spécifie si les paramètres et actions associées doivent être considérés.

Ces classes de services sont les mêmes que celles que nous avons décrites dans les dans les concepts-descriptions (cf. section 4.3.2 page 79). Chacun de ces paramètres peut être redéfini de true vers false. Comme dans un concept-description, nous suggérons non pas ici l'usage d'un paramètre par classe de service, mais plutôt un paramètre unique, ensemble de classes de service.

Cardinality Il exprime la cardinalité du composant-relation sous la forme $1-n$ qui signifie que les relations issues de ce composant-relation peuvent être créées entre 1 description-source et 1 à n descriptions-cibles (n est ∞ ou un entier naturel supérieur ou égal à 1).

Par exemple, un composant-relation d'héritage simple aura une cardinalité $1-1$ (chaque description ne peut hériter que d'une seule description) alors que, pour un héritage multiple, elle sera de $1-\infty$ (chaque description peut hériter de plusieurs descriptions). Nous pouvons ainsi limiter la multiplicité d'un héritage (à $1-3$ par exemple) ce qui revient par exemple à normaliser la programmation au sein d'une équipe de développement ou d'une entreprise.[8]

Précisons que nous parlons bien ici de la cardinalité d'*une* relation instanciée et orientée, allant de sa source (qui déclare la relation) vers sa ou ses cibles (pour un héritage multiple, une classe hérite de plusieurs autres, soit $1-\infty$). Si nous orientons la relation à l'inverse, la cardinalité serait elle aussi inversée (pour un héritage multiple, plusieurs classes peuvent hériter d'une même classe, soit $\infty-1$). Et si nous regardons le graphe complet, il est toujours $\infty-\infty$ à un niveau donné (pour un héritage multiple, un ensemble de classes peuvent hériter d'un ensemble d'autres et être héritées par un troisième ensemble).

Ce paramètre peut être redéfini en remplaçant un entier par un entier inférieur et/ou ∞ par un entier.

Repetition Il indique si une répétition directe de description est permise dans les descriptions-sources et dans les descriptions-cibles des relations issues de ce composant-relation.

C'est une paire de valeurs allowed ou forbidden. Cette valeur n'a de sens que dans le cadre d'une relation avec une cardinalité différente de $1-1$ et seulement du ou des côtés différents de 1. Par exemple, pour une relation avec une cardinalité de type $1-n$, seule la seconde valeur du paramètre Repetition est prise en considération.

Les répétitions indirectes posent des problèmes qu'il est plus simple de résoudre par le système d'assertions que par un simple paramètre. Par exemple, nous

[8] Nous avons choisi de concevoir tous nos composants-relations avec une cardinalité $1-1$ ou $1-n$ mais le modèle supporte également les cardinalités de type $m-n$. Cela peut par exemple être utile pour concevoir un composant-relation décrivant l'*association* en *UML*.

pourrions imaginer un paramètre Indirect_Repetition pour les autoriser ou interdire. Mais, celui-ci, cantonné à un seul composant-relation, ne pourrait pas définir le comportement de l'*implements* de *Java* : une classe peut implémenter plusieurs fois la même interface à la condition *sine qua none* de le faire indirectement. Mais ce dernier « indirectement » fait en réalité référence au graphe de l'héritage inter-interface et non à celui de l'implémentation.

Il peut être redéfini en remplaçant allowed par forbidden pour l'une et/ou l'autre des deux valeurs.

Circularity Il exprime la possibilité de créer des cycles avec des relations issues de ce composant-relation : allowed signifie que des cycles sont permis, forbidden qu'ils sont interdits.

Par exemple, l'héritage et la composition interdisent les cycles alors que l'agrégation les permet le plus souvent.

Ce paramètre peut être redéfini de allowed vers forbidden.

Symmetry Il indique si le composant-relation est symétrique.

Un composant-relation est symétrique s'il fournit lui-même une relation sémantique symétrique. Il est possible d'imaginer ainsi un composant-relation *est-une-sorte-de* pour lequel la description-cible et la description-source sont les termes d'une relation symétrique.

Ce paramètre peut être redéfini de true vers false. Les relations symétriques sont rares. Il est cependant pertinent de pouvoir les signaler car elles peuvent offrir une sémantique opérationnelle particulière (par exemple lors de la liaison dynamique).

Opposite Il indique, s'il existe, le composant-relation inverse.

Un composant-relation est inverse d'un autre (et vice versa), s'ils décrivent une sémantique inverse. Par exemple, un composant-relation de spécialisation est inverse d'un composant-relation de généralisation. Ces définitions entraînent le fait suivant : un composant-relation symétrique est son propre inverse.

Ce paramètre peut être redéfini d'un composant-relation vers none.

Direct_access Ce paramètre permet de préciser si la relation offre un accès direct aux primitives de la description-cible dans la description-source.

Si un accès direct est présent pour une primitive et que cet accès ne présente aucune ambiguïté (cas de la présence d'un autre accès direct ou d'une primitive locale homonyme), alors l'utilisation de la primitive distante peut se faire comme si elle était locale. Naturellement, les relations d'importation privilégient souvent les accès directs. Les valeurs possibles pour ce paramètres sont mandatory, allowed et forbidden.

Il peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Indirect_access Ce paramètre est similaire au précédent mais pour la gestion des accès indirects. Par accès indirect, nous entendons la nécessité de nommer explicitement la description-cible ou l'une de ses instances pour accéder à la primitive.

Par exemple, si Direct_access et Indirect_access valent allowed, deux types d'accès sont possibles. S'ils sont tous deux à forbidden, toutes les primitives de la description-cible sont inaccessibles. Les relations d'utilisation, par définition, ont plutôt tendance à mettre en œuvre des accès indirects.

Précisons que l'accès indirect ne permet pas d'accéder aux primitives masquées (cf. le paramètre Masking ci-dessous).

Ce paramètre peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Polymorphism_implication (spécifique aux relations d'importation) Il indique si la relation implique la mise en place d'un polymorphisme (cf. définition 8 page 51) pour les instances des descriptions liées.

Prenons plusieurs exemples : une simple réutilisation de code interdit le polymorphisme (none), la spécialisation le permet dans le sens source-vers-cible (up), la généralisation dans le sens inverse (down) et nous pouvons imaginer un composant-relation de version qui l'implique dans les deux sens (both).

Ce paramètre peut être redéfini de both vers up, down ou none ou bien de up ou down vers none.

Polymorphism_policy (spécifique aux relations d'importation) Ce paramètre ne prend de sens que si Polymorphism_implication est différent de none. Il précise si le polymorphisme impliquée par la relation, sur les méthodes et sur les attributs (c'est donc une paire de valeurs), doit se faire selon une politique de remplacement (comme c'est le cas pour les attributs en *Java* : hiding) ou de redéfinition (cas des méthodes, toujours en *Java* : overriding).

Ce paramètre ne peut pas être redéfini.

Feature_variance (spécifique aux relations d'importation) Ce paramètre signale le type de variance de la relation par un triplet de valeurs, la première pour les paramètres de méthode, la seconde pour les résultats de fonction, la dernière pour les attributs.

Chaque membre du triplet peut donc être :

- covariant : Le type des paramètres de méthode (ou résultats de fonction ou attributs, cette parenthèse étant valable pour les autres explications) de la description-source doit être identique ou un sous-type[9] du type des paramètres de méthode correspondants dans la description-cible.
- contravariant : Le type des paramètres de méthode de la description-source doit être identique à ou être un sur-type du type des paramètres de méthode correspondants dans la description-cible.
- nonvariant : Le type des paramètres de méthode de la description-source doit être identique à celui des paramètres de méthode correspondants dans la description-cible.
- non_applicable : Aucune contrainte ne s'applique entre le type des paramètres de méthode de la description-source et celui des attributs correspondants dans la description-cible.

Chacune des trois valeurs de ce paramètre peut être redéfinie de covariant ou contravariant vers nonvariant ou non_applicable ou bien de nonvariant vers non_applicable.

Assertion_variance (spécifique aux relations d'importation) Ce paramètre signale le type de variance de la relation pour les assertions. Ils est en fait composé de trois

[9] Si A est source d'une relation R dont B est cible et que le paramètre Polymorphism_implication de R indique up alors A est un sous-type de B. Si Polymorphism_implication est à down, alors B est un sous-type de A. Enfin, si la valeur de ce paramètre est both alors A et B peuvent être considérés comme équivalents du point de vue des types.

valeurs, une première pour les invariants, une deuxième pour les préconditions et une dernière pour les postconditions.

Chacune des valeurs peut être :

– weakened : L'assertion de la description-source doit être identique à ou plus large que celle de la description-cible. C'est-à-dire que la première doit être impliquée par la seconde.
– strengthened : L'assertion de la description-source doit être identique à ou plus restreinte que celle de la description-cible. C'est-à-dire que la première doit impliquer la seconde.
– unchanged : L'assertion de la description-source doit être identique à celle de la description-cible. C'est-à-dire qu'elles doivent être équivalentes.
– non_applicable : Aucune contrainte ne s'applique entre l'assertion de la description-source et celle de la description-cible.

Ce paramètre peut être redéfini de weakened ou strengthened vers unchanged ou non_applicable ou bien de unchanged vers non_applicable.

Dependence (spécifique aux relations d'utilisation) Ce paramètre signale si les instances de la description-cible ont une durée de vie dépendante (dependent) ou non (independent) de celle de la description-source. Le fait d'être dépendant signifie donc que l'instance de la description-cible ne peut *survivre* à celle de la description-source.

Ce paramètre ne peut pas être redéfini.

Sharing_level (spécifique aux relations d'utilisation) Il indique si l'instance de la description-cible peut être partagée (entre parenthèses, la valeur possible du paramètre) :

– par toutes les instances, telle une variable globale générale (global, valeur la plus générale).
– par toutes les instances d'un paquetage, comme une variable globale au sein d'un paquetage (package),
– par toutes les instances d'une même description, telle une instance de classe (ou de description),
– par plusieurs instances, comme une variable d'instance (instance) ou encore
– par une unique instance, donc en tant que variable d'une seule instance (unique_instance, valeur la plus spécifique).

Une seule valeur est possible parmi celles proposées ci-dessus. La mise en œuvre de ce paramètre doit être compatible avec le paramètre de concept-description Sharing_control (cf. section 4.3.2 page 80).

Les redéfinitions sont admises d'une valeur spécifique vers une valeur plus générale.

Read_accessor (spécifique aux relations d'utilisation) Ce paramètre spécifie si l'accès en lecture aux attributs peut se faire directement (optional) ou s'il est nécessaire de passer par des accesseurs (mandatory).

Ce paramètre peut être redéfini de optional vers mandatory.

Write_accessor (spécifique aux relations d'utilisation) C'est le pendant du précédent, pour la modification des attributs : l'affectation d'un attribut est-elle possible directement (optional) ou exclusivement par un accesseur (mandatory) ?

Ce paramètre peut être redéfini de optional vers mandatory.

Adding Ce paramètre indique si l'ajout de primitive est obligatoire (mandatory), permis (allowed) ou interdit (forbidden) au travers de la relation spécifiée. Il

redéfinit de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Notons que la valeur mandatory, pour ce paramètre et pour les suivants, n'a pas vocation à être utilisée fréquemment. Cependant, dans quelques cas, elle peut s'avérer bien utile, comme nous le verrons dans un exemple pour les paramètres Redefining et Effecting.

Ce paramètre peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Removing Il indique si la suppression des primitives est obligatoire (mandatory), permise (allowed) ou interdite (forbidden) au travers de cette relation.

Ce paramètre peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Renaming Ce paramètre précise si le renommage des primitives est obligatoire (mandatory), autorisé (allowed) ou interdit (forbidden) au travers de cette relation.

Ce paramètre peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Redefining (plutôt adapté aux relations d'importation) Ce paramètre indique si la redéfinition des primitives[10] est obligatoire, autorisée ou interdite au travers de cette relation.

Au contraire des précédents, ce paramètre est multi-valué. Nous pouvons donner la valeur mandatory, allowed ou forbidden pour obliger, autoriser ou interdire la redéfinition des assertions, de la signature, du corps et des qualifieurs (visibilité, protection, constance, ...) de la primitive. Pour ce paramètre, la valeur mandatory signifie que toutes les primitives doivent forcément être redéfinies.

Ce paramètre peut être redéfini, chacune des valeurs pouvant l'être de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Masking Ce paramètre est présent pour notifier l'obligation, la possibilité ou l'interdiction de masquer les primitives — et non supprimer (cf. Removing) car elles peuvent être *dé*masquées (cf. Showing) par une autre relation.

Les valeurs mandatory, allowed et forbidden sont valides.

Ce paramètre peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Showing C'est l'inverse de Masking. Il exprime s'il est obligatoire (mandatory), possible (allowed) ou non (forbidden) de rendre de nouveau visible les primitives préalablement masquées.

Il peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Abstracting (plutôt adapté aux relations d'importation) Ce paramètre permet de signifier s'il est obligatoire, possible ou interdit de rendre les primitives abstraites au travers de cette relation (lorsqu'elles sont importées concrètes).

Il peut prendre les valeurs mandatory, allowed ou forbidden.

[10] Notons qu'il serait intéressant de distinguer deux sortes de redéfinition : celles pour lesquelles il est possible de réutiliser tout ou partie de la définition initiale et les autres, pour lesquelles il est nécessaire de tout (re)définir.

Il peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Effecting (plutôt adapté aux relations d'importation) C'est le pendant d'Abstracting, il décrit l'obligation (mandatory), l'autorisation (allowed) ou non (forbidden) de rendre concrètes les primitives abstraites. La valeur mandatory prend ici un intérêt évident pour définir une relation de concrétisation complète.

Il peut être redéfini de mandatory vers allowed ou forbidden ou de allowed vers forbidden.

Adaptation_advice Ce paramètre est multi-valué. Pour chacun des paramètres permettant l'adaptation des primitives (Adding, Removing, Renaming, Redefining, Masking, Showing, Abstracting et Effecting), il indique si cette adaptation est recommandée (advisable), libre (free) ou déconseillée (inadvisable).

Ce conseil ne prend de sens que si la valeur du paramètre correspondant (par exemple Removing) est allowed.

Adaptation_advice peut être redéfini librement.

Le paramètre Adaptation_advice est un peu particulier. Il décrit en effet un conseil méthodologique et non une propriété rigide. Ce genre de paramètre pourrait être développé dans une version ultérieure du modèle *OFL* avec des idées telles le fait de permettre ou déconseiller l'appel de méthode dans les assertions ou les effets de bord dans les fonctions. Il pourrait être mis à profit ou pas selon la volonté du concepteur de langage.

Paramètres pour la persistance

Ci-dessus, nous avons donné les paramètres s'appliquant de manière générale, c'est-à-dire à toute classe de services. Cependant lorsqu'une classe de service est activée, par exemple si nous souhaitons modéliser un langage persistant [72] ou concurrent, des paramètres spécifiques entrent en jeu et doivent être pris en compte. Nous en présentons ci-dessous quelques-uns à titre d'exemples.

Ces paramètres associées aux classes de service seront intégrés au sein du modèle *OFL* par un protocole spécifique qui est actuellement mis au point dans une autre thèse de doctorat réalisée par Adeline Capouillez. Ce travail affinera sans aucun doute les paramètres présentés ci-dessous qui ne le sont donc qu'à titre indicatif.

Transaction_policy (spécifique aux relations d'utilisation) Ce paramètre spécifie la granularité du système de transactions[11] :
- l'application (application) ou
- la méthode (method) ou
- l'instruction (statement) ou
- l'expression (expression) ou
- au plus serré (use) ou
- manuelle, à la charge du programmeur (manual).

Ce paramètre peut être redéfini librement.

[11] Nous profitons de ce paramètre pour d'ores et déjà donner l'opportunité d'imaginer les possibilités de politiques de composition de relations. Si une relation indique *method* et l'autre *instruction*, nous pourrions choisir de composer en prenant la plus sévère (*instruction*) ou bien la plus lâche (*method*) ce qui donnera deux sémantiques de composition bien différentes.

Object_loading (spécifique aux relations d'utilisation) Ce paramètre spécifie à quel moment un objet persistant doit être chargé en mémoire volatile.

Par exemple, la politique choisie peut être d'effectuer ce chargement :
- au début des applications utilisant cet objet (application) ou
- au début des méthodes utilisant cet objet (method) ou
- au début des instructions utilisant cet objet (statement) ou
- au début des expressions utilisant cet objet (expression) ou
- au plus tard, quand cela est indispensable (use) ou
- manuellement, à la charge du programmeur (manual).

Ce paramètre peut être redéfini librement.

Load_check (spécifique aux relations d'utilisation) Ce paramètre permet de spécifier quels contrôles doivent être réalisés lors du chargement d'un objet persistant vers la mémoire volatile :
- aucun, à la charge du programmeur (none) ou
- simple vérification de la disponibilité, due à un précédent chargement, de l'objet en mémoire (object) ou
- si le test précédent a échoué, vérification de la disponibilité en mémoire du type de l'objet à charger (type) ou
- si le test précédent a échoué, vérification de la disponibilité en mémoire d'un type *compatible* avec celui de l'objet à charger (compatible_type) ou
- si le test précédent a échoué, un adaptateur [12] est-il disponible pour cet objet (adapter) ?

Ce paramètre peut être redéfini selon toutes les combinaisons allant de la plus élaborée (adapter) vers la plus simple (none) : nous pouvons donc redéfinir compatible_type en type mais pas l'inverse.

Failed_loading (spécifique aux relations d'utilisation) Il indique quelle doit être la réaction à un échec de chargement d'objet. Cela peut être :
- aucune, à la charge du programmeur (none) ou
- simple notification à l'application (notification) ou
- de plus si cela s'avère insuffisant, annulation de la transaction courante (transaction) ou
- de plus si cela s'avère insuffisant, lancement d'une procédure de réparation (repair) ou
- de plus si cela s'avère insuffisant, arrêt de l'application (lethal).

Ce paramètre peut être redéfini selon toutes les combinaisons allant de la plus élaborée (lethal) vers la plus simple (none).

Object_updating (spécifique aux relations d'utilisation) Ce paramètre spécifie à quel moment la représentation persistante d'un objet est mise à jour par rapport à sa *copie* volatile.

Les valeurs possibles sont les correspondantes de celles d'Object_loading :
- à la fin des applications utilisant cet objet (application) ou
- à la fin des méthodes utilisant cet objet (method) ou
- à la fin des instructions utilisant cet objet (statement) ou
- à la fin des expressions utilisant cet objet (expression) ou
- au plus tôt, dès que cela est possible (use) ou
- manuellement, à la charge du programmeur (manual).

Ce paramètre peut être redéfini librement.

Failed_updating (spécifique aux relations d'utilisation) C'est le pendant, pour la

mise à jour, de failed_loading. Les valeurs envisageables sont identique ainsi que les possibilités de redéfinitions.

...

Paramètres pour la concurrence

Object_locking (spécifique aux relations d'utilisation) Ce paramètre exprime le moment où l'objet utilisé doit être verrouillé, dans le but d'éviter des accès concurrents, sources de problèmes bien connus.

Il est possible d'effectuer le verrouillage :
- pour les applications utilisant cet objet (application) ou
- pour les méthodes utilisant cet objet (method) ou
- pour les instructions utilisant cet objet (statement) ou
- pour les expressions utilisant cet objet (expression) ou
- au plus juste, au moment de l'utilisation de l'objet (use) ou
- manuellement, à la charge du programmeur (manual).

Ce paramètre peut être redéfini librement.

...

4.4.3 Assertions

Comme pour les composants-descriptions, il est possible d'associer à chaque composant-relation un ensemble d'invariants qui définissent et contrôlent son utilisation. Ces assertions peuvent s'appliquer à un composant-relation particulier (c'est le cas de la première assertion ci-dessous) ou être structurelles (comme la seconde assertion ci-dessous), c'est-à-dire utiles pour toutes instances de concept-relation.

Nous pouvons ainsi donner comme exemples d'assertions :

1. Nous voulons, pour un composant-relation, vérifier une covariance stricte sur les paramètres de méthode. Voici l'invariant que nous pourrions exprimer. « Soient D1, D2 et D3 des descriptions, CR1 un composant-relation d'importation pour lequel Feature_variance pour les paramètres de méthode est égal à covariant et CR2 un composant-relation d'utilisation, si D1 est source d'une instance de CR1 dont D2 est cible et D2 est source d'une instance de CR2 dont D3 est cible, alors l'instance de CR2 équivalente dont D1 est source ne doit pas avoir D3 pour cible. » Il suffit en effet de tester que les utilisations mènent à des descriptions différentes pour vérifier la covariance stricte, le contrôle de la covariance large étant normalement effectué par un invariant structurel[12] s'appliquant à tout langage.
2. Le second invariant qui nous servira d'exemple interdit la répétition directe de cible pour les composants-relations dont le paramètre Repetition le spécifie. « Soient D1 et D2 des descriptions et CR un composant-relation pour lequel Repetition est égal à forbidden pour les descriptions-cibles, alors D1 ne peut pas être plus d'une fois cible directe d'une instance de CR ayant pour source D2. »

[12] ou par une action de contrôle : cf. l'action verify_variance section 6.6 page 134

Nous voyons donc, qu'à l'aide d'assertions et principalement d'invariants, il est possible de spécifier des contraintes très fines pour les comportements de concept-relation, contraintes qui définissent des cas particuliers d'usage qu'il ne seraient pas forcément raisonnable de faire apparaître au niveau des paramètres.

4.4.4 Bilan sur les concepts-relations

Les tableaux 4.3 page ci-contre et 4.4 page 96 résument la composition d'un concept-relation entre descriptions.

4.5 Concept-langage

Un concept-langage est un méta-méta-langage, chacune de ses instances, nommée *composant-langage*[16], représente une notion importante et intuitive. Elle décrit, comme son nom l'indique, un langage modélisé et son rôle principal est de fédérer des composants-descriptions et composants-relations. À terme, cette modélisation des langages devra permettre de mettre en œuvre et de contrôler l'interopérabilité entre objets engendrés par des composants-langages différents.

4.5.1 Données

Nous rappelons que, dans cette section, nous introduisons des éléments de réification qui seront plus largement présentés dans la section 5.2 page 108.

Les informations de réification, associées à un composant-langage, sont :

– Chaque composant-langage est constitué de plusieurs constituants :

1. un ensemble de composants-descriptions (ceux qui sont valides pour ce composant-langage) et
2. un ensemble de composants-relations (ceux qui sont valides pour ce composant-langage), avec la condition que, pour chaque composant-relation, au moins un composant-description sélectionné soit compatible en tant que source et un autre (ou le même) en tant que cible.
3. un ensemble de triplets <composants-relations, composant-description_source, composant-description_cible> donnant tous les appariements valides entre composant-description_source et composant-description_cible pour un composant-relation. Par exemple pour Java, signalons entre autres <Implementation, Class, Interface>.

L'ensemble des composants-relations peut, soit être déduit de l'ensemble de composants-descriptions, soit être donné explicitement, ce qui permet éventuellement de réduire, à ce niveau, les spécifications de compatibilité décrites dans chaque composant-description. Par exemple, un programmeur peut souhaiter utiliser un langage à la *Eiffel* sans autoriser la clientèle expansée. Évidemment, seule une restriction des spécifications peut être effectuée, pas une extension.
De la même manière, il est possible de déduire les ensembles de composants-descriptions et de composants-relations de l'ensemble de triplets.

[13]Ce paramètre est spécifique aux relations d'importation.

[14]Ce paramètre est spécifique aux relations d'utilisation.

[15]Les propriétés spécifiques à chaque relation sont développées et complétées dans la section 5.2.9 page 115.

[16]Nous verrons, dans la section 5.1 page 103, une vue globale de l'architecture du modèle.

Informations générales au modèle *OFL*		
Assertion	**Type**	**Exemple de valeur**
Structural invariants	set of condition	cf. page 93
Action	**Type**	**Exemple de valeur**
Actions	set of routine	*cf. page 117*

Informations spécifiques au concept-relation		
Élément de réification	**Type**	**Exemple de valeur**
Extension	set of relationship-component	{Inheritance, Clientèle}

Informations spécifiques à chaque composant-relation		
Élément de réification	**Type**	**Exemple de valeur**
Language	language	Java-like
Valid source description-concepts	set of description-concept	{Class}
Valid target description-concepts	set of description-concept	{Class, Interface}
Valid Qualifiers	set of <atom_name, set of string>	{<relationship, {"in"}>}
Extension	set of relationship	{Inheritance1, Inheritance2}
Paramètre	**Type**	**Exemple de valeur**
Name	string	"Specialisation"
Kind	*import* | *use* | *type-object* | *objects*	import
Context	*language* | *library*	library
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Cardinality	<integer, integer>	<1, ∞>
Repetition	<*allowed* | *forbidden*, *allowed* | *forbidden*>	<forbidden, forbidden>
Circularity	*allowed* | *forbidden*	forbidden
Symmetry	boolean	false
Opposite	*none* | relationship-concept	none
Direct_access	*mandatory* | *allowed* | *forbidden*	mandatory
Indirect_access	*mandatory* | *allowed* | *forbidden*	forbidden
Polymorphism_-implication[13]	*none* | *up* | *down* | *both*	up

à suivre dans le tableau 4.4 page suivante

TAB. 4.3 – Récapitulatif de la structure de concept-relation entre descriptions (1/3)

suite du tableau 4.3 page précédente

Informations spécifiques à chaque composant-relation		
Paramètre	**Type**	**Exemple de valeur**
Polymorphism_policy[13]	<*hiding* | *overriding*, *hiding* | *overriding*>	<overriding, hiding>
Feature_variance[13]	<*covariant* | *contravariant* | *nonvariant* | *non_applicable*, *covariant* | *contravariant* | *nonvariant* | *non_applicable*, *covariant* | *contravariant* | *nonvariant* | *non_applicable*>	<covariant, nonvariant, non_applicable>
Assertion_variance[13]	<*weakened* | *strengthened* | *unchanged* | *non_applicable*, *weakened* | *strengthened* | *unchanged* | *non_applicable*, *weakened* | *strengthened* | *unchanged* | *non_applicable*>	<strengthened, weakened, strengthened>
Dependence[14]	*dependent* | *independent*	independent
Sharing_level[14]	*global* | *package* | *description* | *instance* | *unique_instance*	instance
Read_accessor[14]	*optional* | *mandatory*	optional
Write_accessor[14]	*optional* | *mandatory*	optional
Adding	*mandatory* | *allowed* | *forbidden*	allowed
Removing	*mandatory* | *allowed* | *forbidden*	forbidden
Renaming	*mandatory* | *allowed* | *forbidden*	allowed
Masking	*mandatory* | *allowed* | *forbidden*	forbidden
Showing	*mandatory* | *allowed* | *forbidden*	forbidden

à suivre dans le tableau 4.5 page ci-contre

TAB. 4.4 – Récapitulatif de la structure de concept-relation entre descriptions (2/3)

suite et fin du tableau 4.4 page précédente

Informations spécifiques à chaque composant-relation		
Paramètre	**Type**	**Exemple de valeur**
Redefining	<*mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*>	<allowed, forbidden, allowed, forbidden>
Abstracting	*mandatory* | *allowed* | *forbidden*	forbidden
Effecting	*mandatory* | *allowed* | *forbidden*	forbidden
Adaptation_advice	<*Adding* : *advisable* | *free* | *inadvisable*, *Removing* : *advisable* | *free* | *inadvisable*, *Renaming* : *advisable* | *free* | *inadvisable*, *Redefining* : *advisable* | *free* | *inadvisable*, *Masking* : *advisable* | *free* | *inadvisable*, *Showing* : *advisable* | *free* | *inadvisable*, *Abstracting* : *advisable* | *free* | *inadvisable*, *Effecting* : *advisable* | *free* | *inadvisable*>	<*Adding* : *advisable*, *Removing* : *inadvisable*, *Renaming* : *advisable*, *Redefining* : *free*, *Masking* : *inadvisable*, *Showing* : *inadvisable*, *Abstracting* : *inadvisable*, *Effecting* : *free*>
Assertion	**Type**	**Exemple de valeur**
Invariants	set of condition	*cf. exemples page 93*

Informations spécifiques à chaque relation[15] (cf. section 5.2.9 page 115)		
Élément de réification	**Type**	**Exemple de valeur**
Source descriptions	set of type	{Class23, Class28}
Target descriptions	set of type	{Interface1, Class22}
Removed features	set of feature	{a_feature}
Renamed features	set of <feature, feature_name>	{<a_feature, its_new_name>}
Redefined features	set of <feature, feature>	{<a_feature, its_new_version>}
Hidden features	set of feature	{another_feature}
Shown features	set of feature	{a_feature}
Abstracted features	set of feature	{a_feature_again}
Effected features	set of feature	{another_feature}

TAB. 4.5 – Récapitulatif de la structure de concept-relation entre descriptions (3/3)

Enfin, chaque concept-langage et chacune de ses instances (les composants-langages) a connaissance de son extension. C'est-à-dire que chaque concept-langage a accès à la liste des méta-langages qu'il décrit et chaque composant-langage possède l'ensemble des langages qu'il représente. Précisons que souvent, mais ce n'est pas une obligation, un composant-langage est un singleton et ne représente donc qu'un seul langage.

4.5.2 Paramètres

Chaque composant-langage dispose, comme les autres instances de concept de notre modèle, d'un certain nombre de paramètres qui représentent les principales possibilités d'adaptation du comportements d'un langage. En voici quelques exemples :

Name Il s'agit simplement du nom du composant-langage, par exemple "MyLanguage" ou "Java-like".

Persistence Ce paramètre booléen indique si le langage, ses composants-descriptions et ses composants-relations doivent intégrer et tenir compte des paramètres et actions concernant la *classe de services* persistance.

Concurrency Il indique, comme le précédent, l'activation d'une classe de services, ici la concurrence.

... Pour chaque classe de services, nous avons donc un paramètre booléen qui spécifie si les paramètres et actions associées doivent être considérés.

La liste des classes de service est donnée ici à titre indicatif, elle peut évidemment être amenée à intégrer de nouvelles classes de service au fur et à mesure de leur conception. Nous envisageons donc, lors de l'implantation d'*OFL* d'utiliser un paramètre constitué d'un ensemble de classes de service plutôt qu'un paramètre par classe de service.

4.5.3 Redéfinitions

Dans l'idée de permettre l'utilisation de composants en réduisant explicitement leurs capacités, nous pouvons imaginer, par exemple, d'utiliser une instance de concept-relation, un composant-relation (la même chose peut se faire, nous l'avons vu avec un composant-description) en limitant ses capacités. Nous pouvons pour cela définir une liste formée de triplets <composant-relation, nom_paramètre, valeur> dans laquelle il est possible de spécifier une nouvelle valeur pour un paramètre de concept-relation. Il est ici aisé de comprendre qu'il faut s'en tenir à limiter les capacités du composant-relation et en aucun cas les augmenter.

Les valeurs des paramètres peuvent être redéfinies (sous certaines conditions, cf. section 2.2 page 23) par le composant-langage, mais elles ne sont pas les seules. Nous trouvons en effet dans les composants-descriptions et les composants-relations un système de *qualifieurs* — dont la sémantique est gérée par l'intermédiaire des assertions. Un qualifieurs est un mot-clé du langage qui qualifie une entité. Par exemple en *Java*, une méthode peut être qualifiée de private, final, static, ... De même, une classe peut être public, abstract, ... Ce mécanisme de qualifieurs permet en particulier d'augmenter l'expressivité du modèle *OFL* en l'étendant à des entités non paramétrables.

Des éléments de réification de concepts-descriptions et concepts-relations définissent, pour cet usage, comme nous l'avons vu, les listes de qualifieurs valides pour

une description, un attribut, une méthode, un paramètre de méthode, une assertion ou encore une relation. Il est possible dans un composant-langage de réduire ces listes (toujours donc sur le principe du *qui peut le plus peut le moins*) par redéfinition. Nous modélisons une telle redéfinition par un ensemble de triplets <composant-description, entité_concernée[17], liste_valeurs> ou <composant-relation, entité_concernée, liste_valeurs>, liste_valeurs étant un sous-ensemble de la liste initiale de valeurs permises.

4.5.4 Assertions

À chaque concept-langage, il est possible d'associer deux ensembles d'invariants :

1. **La vérification de contraintes d'utilisation des concepts-descriptions et concepts-relations associés au langage.** Il s'agit donc d'assertions pour le méta-programmeur, chargées de préciser la sémantique du langage modélisé. De ce premier cas, nous pouvons donner les deux exemples suivants :
   - Nous souhaitons vérifier que, dans une application écrite dans ce langage, toute relation est posée concomitamment à son opposée (dans l'autre sens), pour peu qu'elle en possède une[18]. Par exemple, lorsque nous posons une relation de spécialisation entre une description-source et une description-cible, nous devons également établir une généralisation dans l'autre sens. Une expression en français de cet invariant peut être « Soient D1 et D2 des descriptions et CR1 et CR2 des composants-relations opposés, si D1 est source d'une instance de CR1 dont D2 est cible alors D1 doit être cible d'une instance de CR2 dont D2 est source. »
   - Comme second exemple, nous choisissons de vérifier la règle qu'aucune circularité ne doit être directe. Ce choix peut s'exprimer simplement ainsi : « Soient D1 et D2 des descriptions et CR un composant-relation, si D1 est source d'un CR direct dont D2 est cible alors D1 ne peut pas être cible d'un CR direct dont D2 est source. » Notons que cette condition s'applique naturellement et sans précision supplémentaire aux relations non circulaires.
2. **De manière plus générale, le contrôle de contraintes structurelles.** Il s'agit en fait d'assertions décrites pour tout concept-langage et présentées ici pour simplifier l'explication. Ce sont donc des assertions pour le concepteur d'*OFL*, chargées de préciser la sémantique du modèle et de tous les langages qu'il modélise. Nous présentons deux exemples de tels contrôles. Ces exemples tendent à vérifier que les valeurs de certains paramètres sont correctement prises en considération par le système.
   - Cet invariant vérifie que deux relations opposées ne peuvent être posées concomitamment (dans le même sens) : une description, par exemple, ne peut en même temps en spécialiser et en généraliser une même autre. Le paramètre qui donne lieu à ce contrôle est ici Opposite (cf. section 4.4.2 page 87). « Soient D1 et D2 des descriptions et CR1 et CR2 des composants-relations opposés, si D1 est source d'une instance de CR1 dont D2 est cible, alors D1 ne peut pas être source d'une instance de CR2 dont D2 est cible. »

[17] entité_concernée est donc description,attribute, method, parameter ou relationship.

[18] Attention ! Il s'agit bien ici d'une assertion, c'est-à-dire d'un test qui peut être activé ou pas selon la confiance que nous possédons dans le code à vérifier. Il ne s'agit pas d'un processus automatique de placement d'une relation opposée.

– Nous souhaitons également vérifier que, quelque soit le langage décrit, aucun cycle n'est réalisé pour des composants-relations dont le paramètre Circularity est égal à forbidden. « Soient D1 et D2 des descriptions et CR un composant-relation pour lequel Circularity est égal à forbidden, si D1 est source d'une instance de CR dont D2 est cible, alors D1 ne peut pas être cible d'une instance CR dont D2 est source. »

4.5.5 Exemples

Pour poursuivre dans la lignée de nos exemples sur *Eiffel* et *Java*, nous pouvons dire que le composant-langage *Eiffel* est constitué :

1. des neuf composants-descriptions d'*Eiffel* : classe (cf. page 61), classe abstraite (cf. page 62), classe générique (cf. page 62), classe abstraite générique (cf. page 63), classe expansée (cf. page 63), classe expansée générique (cf. page 63), type fondamental (cf. page 64), type binaire (cf. page 64) et néant (cf. page 65) et
2. des sept composants-relations d'*Eiffel* : héritage (cf. page 51), clientèle (cf. page 52), clientèle expansée (cf. page 53), clientèle générique (cf. page 54), clientèle générique expansée (cf. page 54), généricité (cf. page 54) et généricité expansée (cf. page 55).

Pour *Java*, le composant-langage est construit de manière exactement similaire. Il est donc constitué :

1. des treize composants-descriptions de *Java* : classe (cf. page 65), classe abstraite (cf. page 66), interface (cf. page 66), classe membre statique (cf. page 67), classe membre statique abstraite (cf. page 67), classe membre (cf. page 67), classe membre abstraite (cf. page 68), classe locale (cf. page 68), classe locale abstraite (cf. page 69), classe anonyme (cf. page 69), interface membre statique (cf. page 69), tableau (cf. page 70) et type primitif (cf. page 70)
2. et des huit composants-relations de *Java* : héritage inter-classe (cf. page 55), héritage inter-interface (cf. page 56), concrétisation (cf. page 57), implémentation (cf. page 57), agrégation (cf. page 58), agrégation de classe (cf. page 59), composition (cf. page 59) et composition de classe (cf. page 60).

4.5.6 Bilan sur les concepts-langages

Le tableau 4.6 page ci-contre réalise une synthèse de la structure d'un concept-langage (et donc de ses instances, les composants-langages)[19]. Rappelons que la seconde catégorie d'assertions, celles qui sont structurelles et s'appliquent donc à tout langage, n'est pas réellement encapsulée dans l'entité concept-langage ; mais nous les avons présentées en même temps que les autres car les sortir de leur contexte d'utilisation nuirait à la lecture.

[19]Les propriétés spécifiques à chaque langage sont développées dans la section 5.2.7 page 114.

Informations générales au modèle *OFL*		
Assertion	**Type**	**Exemple de valeur**
Structural invariants	set of condition	cf. exemples section 2 page 99

Informations spécifiques au concept-langage		
Élément de réification	**Type**	**Exemple de valeur**
Extension	set of language-component	{MJava, MEiffel}

Informations spécifiques à chaque composant-langage		
Élément de réification	**Type**	**Exemple de valeur**
Valid description-components	set of description-component	{Class, Interface}
Valid relationship-components	set of relationship-component	{Inheritance, Implementation, Aggregation}
Valid relationships	set of <relationship-component, description-component, description-component>	{<Implementation, Class, Interface>}
Extension	set of language	{Java, Eiffel, CPP}
Paramètre	**Type**	**Exemple de valeur**
Name	string	"Java-like"
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Redéfinition	**Type**	**Exemple de valeur**
Description limitations	set of <description-component, parameter-name, value>	{<cd, Generator, false>}
Relationship limitations	set of <relationship-component, parameter-name, value>	{<cr1, Circularity, forbidden>, <cr2, Polymorphism_implication, none>}
Valid-Qualifiers redefinitions	set of <description-component | relationship-component, atom_name, set of string>	{<cd1, method, {"public", "private"}>, <cd2, attribute, {"static", "final"}>}
Assertion	**Type**	**Exemple de valeur**
Local invariants	set of condition	cf. exemples section 1 page 99

TAB. 4.6 – Récapitulatif de la structure de concept-langage

Chapitre 5

Architecture du modèle *OFL*

5.1 Vue d'ensemble d'*OFL*

Nous avons présenté les trois principaux concepts qui permettent d'adapter la sémantique opérationnelle des langages à objets à classes :

1. les concepts-langages,
2. les concepts-descriptions et
3. les concepts-relations.

Il nous faut maintenant donner une vision plus large du modèle en y intégrant ces trois concepts. La figure 5.1 page suivante[1] illustre l'utilisation du modèle *OFL* pour décrire une application. Nous montrons dans cette figure les trois niveaux de modélisation nécessaires :

1. le niveau application regroupe les descriptions et les objets des applications dont le langage est décrit en *OFL* (*OFL-instances* et *OFL-données*),
2. le niveau langage décrit les composants du langage de programmation (*OFL-composants*) et
3. le niveau *OFL*-cœur représente la réification de ces composants : les *OFL-concepts* — dont font partie les concepts-langages, concepts-descriptions et concepts-relations — et les *OFL-atomes*.

Nous en profitons pour ajouter les définitions d'*OFL*-instance (cf. définition 15) et d'*OFL*-donnée (cf. définition 16).

Définition 15 (*OFL*-instance) *Une OFL-instance est une instance d'OFL-composant.* Les descriptions, les relations inter-description, les méthodes, les objets, *etc.* d'une application dont le langage est décrit en *OFL* sont des *OFL*-instances.

Définition 16 (*OFL*-donnée) *Une OFL-donnée est une instance d'OFL-instance. Les OFL-données sont les instances de description et constituent les données de l'application.*

[1] Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48. Les identificateurs sont donnés en français sur la figure 5.1 page suivante pour améliorer sa lisibilité bien que dans la section sur les atomes (section 5.1.3 page 106), ils prennent leur forme anglaise (exemple : Relation dans la figure = Relationship dans la section sur les atomes).

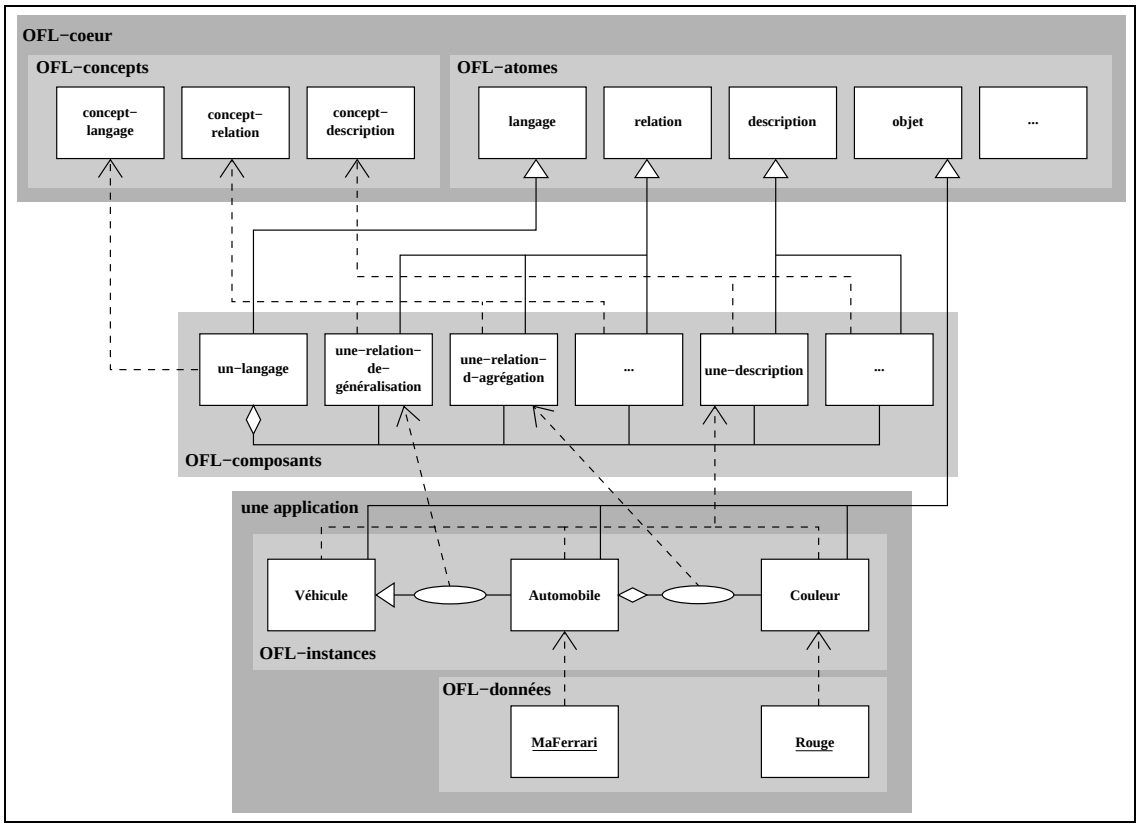

FIG. 5.1 – Exemple d'utilisation d'*OFL*

5.1.1 Le niveau application

Pour décrire une application, le programmeur utilise les services offerts par le niveau langage. Il crée des *OFL*-instances, qui sont les descriptions et les relations de son application, par instanciation des *OFL*-composants. À l'exécution, les objets de l'application, nommés *OFL*-données, sont des instances des *OFL*-instances représentant les descriptions.

Les *OFL*-instances

Chaque description ou relation décrite par le programmeur est modélisée par une *OFL*-instance. La figure 5.1 page ci-contre propose un exemple d'application qui comprend cinq *OFL*-instances :
- trois descriptions : Véhicule, Automobile et Couleur,
- une relation de spécialisation : Automobile *hérite* de Véhicule et
- une relation d'agrégation : Automobile *a un attribut* de type Couleur.

Les *OFL*-données

Dans l'application, chaque instance de description est modélisée à l'exécution par une *OFL*-donnée. La figure 5.1 page précédente en présente deux :
- MaFerrari, instance de la description Automobile et
- Rouge, instance de la description Couleur.

Remarquons que les *OFL*-instances qui représentent des descriptions spécialisent l'*OFL*-atome objet. En effet, objet est la réification des données de l'application (les *OFL*-données) et constitue donc la racine de l'arbre de spécialisation des *OFL*-instances représentant des descriptions.

5.1.2 Le niveau langage

Le niveau langage décrit les différentes sortes de relations et de descriptions qu'il est possible d'utiliser dans le langage modélisé. Les relations sont des instances de concept-relation, les descriptions des instances de concept-description. Le langage lui-même est une instance de concept-langage. Il a pour principale fonction de regrouper les relations et descriptions qu'il met à la disposition du programmeur.

Les *OFL*-composants

Le niveau langage n'est constitué que d'*OFL*-composants. La figure 5.1 page ci-contre recense :
- des composants-descriptions dont une-description,
- des composants-relations dont une-relation-de-spécialisation et une-relation-d-agrégation et
- un composant-langage un-langage.

Il est possible de se représenter un composant-description sous la forme d'une méta-classe, un composant-relation comme une méta-relation et, de la même manière un composant-langage comme un méta-langage.

5.1.3 Le niveau *OFL*-cœur

Le niveau *OFL*-cœur (cf. définition 9 page 74) permet de paramétrer les langages de programmation (niveau langage) et donc les programmes (niveau application). Nous avons choisi (cf. chapitre 4 page 73) de paramétrer les trois notions essentielles : les relations, les descriptions et les langages. Cependant, il est nécessaire de réifier bien d'autres composants, tels les objets, les méthodes, les assertions, etc. pour modéliser complètement un langage. Le niveau *OFL*-cœur contient donc deux sortes d'entités :

1. les *OFL*-concepts qui décrivent la partie paramétrable des relations, descriptions et langages : c'est dans les *OFL*-concepts que sont définis les paramètres hyper-génériques, et
2. les *OFL*-atomes qui décrivent la partie non-paramétrable de ces trois concepts ainsi que tous les autres composants : les *OFL*-atomes apportent donc la réification indépendamment du paramétrage.

Les *OFL*-concepts

Ce sont donc les entités que nous avons décrites dans les sections précédentes. La figure 5.2 page suivante[2] montre l'intégralité de la classification des *OFL*-concepts. Rappelons que les *OFL*-concepts sont les seules entités paramétrées dans notre modèle.

La tâche du méta-programmeur consiste à créer un *OFL*-composant, instance d'*OFL*-concept, en donnant une valeur à chacun des paramètres décrit dans ce dernier. Il définit par ce biais le comportement de chaque future instance de l'*OFL*-composant. Si les actions prévues n'offrent pas au méta-programmeur la sémantique opérationnelle qu'il souhaite associer à un *OFL*-composant, il doit alors modifier le code de ces actions. Cette possibilité laisse le modèle *OFL* ouvert mais ne doit être utilisée que dans des contextes très spécifiques. En effet, le travail du méta-programmeur est dans ce cas beaucoup plus lourd que la simple valuation de paramètres.

Les *OFL*-atomes

Les *OFL*-atomes représentent la réification (hors paramètres hyper-génériques) des entités du modèle. La figure 5.3 page 108[3] illustre une partie de ces *OFL*-atomes[4].

[2] Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48.

[3] Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48.

[4] Les éléments ont été traduits en français sur la figure alors que les identificateurs choisis sont en fait en anglais, comme vous le verrez. Ce choix a été fait pour rendre la figure plus agréable, la traduction ne faisant apparaître aucune ambiguïté notable.

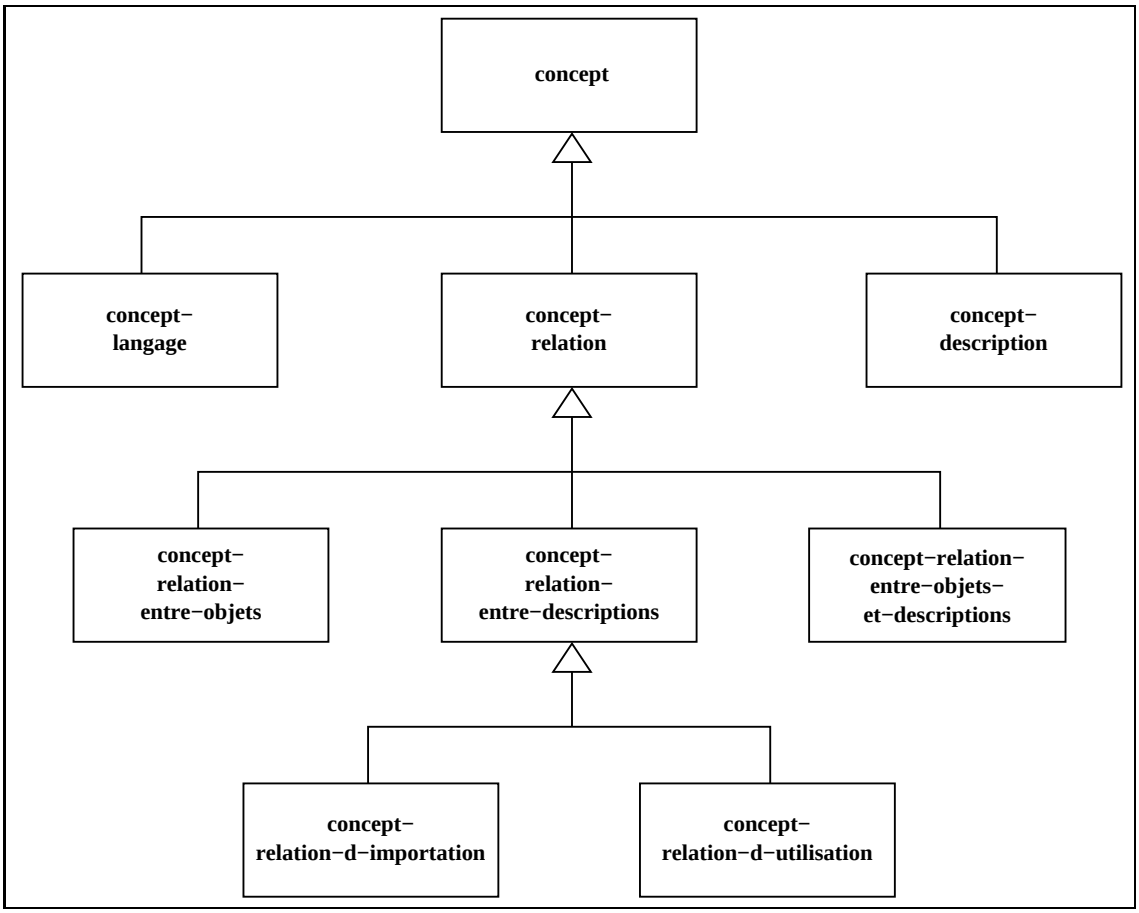

FIG. 5.2 – Les *OFL*-concepts

Les relations, descriptions et langages possèdent également leur *OFL*-atome qui décrit la partie de leur structure et de leur comportement qui n'est pas paramétrée. Sur la figure 5.1 page 104 nous pouvons par exemple noter que l'*OFL*-composant une-relation-d-agrégation est une spécialisation de l'*OFL*-atome relation.

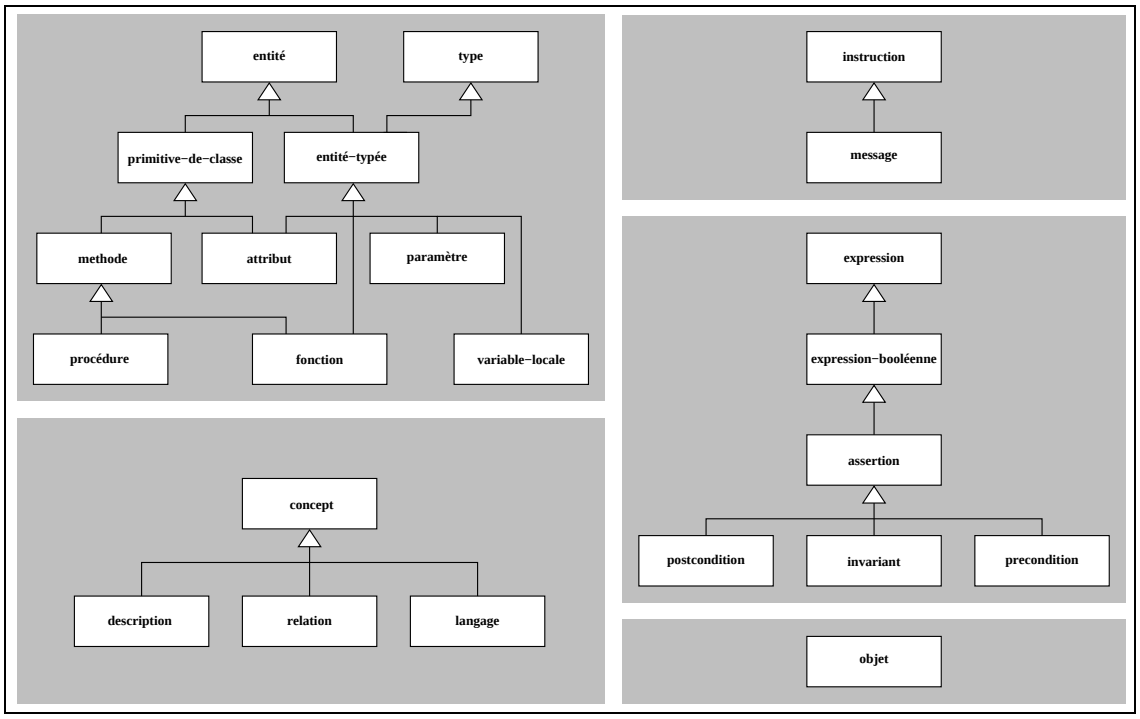

FIG. 5.3 – Les *OFL*-atomes

Dans une application par exemple, toutes les primitives de description sont instances d'un descendant de primitive, les expressions sont instances d'expression ou d'un de ses descendants et tous les objets sont instances de objet. *OFL* offre ainsi une réification complète des entités présentes à l'exécution d'une application.

Notez qu'*a priori* nous ne souhaitons pas réifier complètement les instructions contenues dans les applications. Nous préférons, au moins dans un premier temps, faire tout simplement usage d'un langage connu pour celles-ci.

5.2 Éléments d'implémentation des atomes

Les atomes (*OFL*-atomes sur la figure 5.1 page 104) présentent la structure des éléments du modèle *OFL*. Il s'agit donc de la réification des principales entités nécessaires à une application décrite dans un langage à objets à classes. Nous allons maintenant donner un aperçu de la définition de ces atomes en commentant nos choix.

Précisons que cet aperçu est donné dans le but de faciliter la compréhension de ces atomes. D'autres choix d'implémentation peuvent, s'ils sont équivalents, cependant être faits. De plus, il faut être conscient que chacune des propriétés d'atome que nous proposons ci-dessous peut, soit être une simple valeur stockée (un attribut par exemple), soit le résultat d'un calcul (le résultat d'une fonction par exemple).

5.2.1 Atome Object

Il est la réification de la structure commune à tous les objets finaux des applications décrites en *OFL*. Dans la figure 5.1 page 104 par exemple, ces objets sont MaFerrari, instance directe de Automobile, et Rouge, instance directe de Couleur.

Sa structure se compose de :

- identifier : URI ; Chaque objet possède un identifiant unique. En prévision de la gestion d'applications distribuées, le type URI a été sélectionné. Un URI (*Uniform Resource Identifier*) est une adresse référençant de manière unique une entité (une page, un service, un objet, ...) sur Internet[5]. En première approche, le type integer peut se substituer à URI pour les applications non distribuées.
- instanciation : instanciationRelationship ; Tout objet d'une application est une instance d'une description (souvent une classe) de cette application. instanciation signale donc la relation qui lie l'objet à sa description directe. Cette relation est bien connue en programmation orientée-objets puisqu'il s'agit de l'instanciation. L'objet courant en est la source, sa description la cible. Comme nous l'avons déjà indiqué, l'instanciation n'est pas au cœur des préoccupations de cette thèse qui se concentre plutôt sur les relations inter-descriptions.
- fields : set of Object ; Chaque objet possède des champs décrits par les attributs de sa description. fields est l'ensemble des objets formant ces champs.

5.2.2 Atome Feature

Feature est l'atome réifiant une primitive (cf. définition 17) de description.

Définition 17 (primitive) *Une primitive est une partie de la définition donnée par une description (cf. définition 3 page 47), qu'il s'agisse d'un état ou d'un comportement. Les primitives sont couramment les attributs, les méthodes, les constructeurs, etc. d'une description.*

Feature est structuré ainsi :

- description : Description ; Toute primitive est définie dans une description[6]. Si une description a connaissance de chacune de ses primitives (cf. section 5.2.8 page 114), le contraire est également exact. description est donc la description qui encapsule la primitive[7].
- name : string ; Il s'agit tout simplement du nom de la primitive.
- qualifiers : set of string ; La définition des primitives d'une description peut être précisée par un certain nombre de qualifieurs. Par exemple (mais ce n'est qu'un exemple), en *Java*, une primitive peut être précédée de final, private, static, ... De tels qualifieurs sont étroitement dépendants du langage de programmation. qualifiers est la liste de ceux qui sont *actifs* pour la primitive courante. Le métaprogrammeur décrivant un nouveau langage spécifie, dans le concept-description ou le concept-langage correspondant, la liste des qualifieurs permis pour chacun des atomes concernés (cf. section 4.5 page 94). Il utilise les assertions du concept-description ou du concept-langage pour contrôler l'utilisation de chacun de ces qualifieurs.

[5] Le lecteur non familier de cette notion trouvera toutes les explications nécessaires sur le Web à l'URI « http://www.w3.org/Addressing/ ».

[6] Cela est à remettre en cause dans le cas d'un langage qui n'encapsule pas ses primitives, tel *CLOS*.

[7] Cette propriété est ignorée pour les primitives non encapsulées, comme les méthodes en *CLOS*.

Atome Attribute

Attribute hérite de Feature puisqu'un attribut est une primitive. Il lui ajoute la structure suivante :

- use : UseRelationship ; Un attribut d'une description (la source) définit une relation d'utilisation vers la description décrivant son type (la cible). use est cette relation.
- isDescriptionAttribute : boolean ; Cette propriété permet de différencier les attributs de descriptions (ou *variables de classe*) de ceux d'instances. Il prend la valeur true dans le premier cas, false dans le second. Il est également possible de proposer une séparation des types descriptionAttribute et instanceAttribute, cela pouvant relever d'un choix d'implémentation.
- isConstant : boolean ; true si l'attribut est constant (sa valeur ne peut être modifiée après initialisation), false s'il s'agit d'une variable.

Aussi bien isDescriptionAttribute que isConstant pourraient être gérés par la liste de qualifieurs (nous aurions alors un qualifieur signifiant *attribut de description* et un autre pour *constant*). Cependant, nous avons choisi de mettre en exergue les capacités les plus fréquentes des primitives réifiées pour faciliter leur traitement et de laisser aux soins des qualifieurs les capacités plus marginales. Par ce choix pragmatique, nous permettons une utilisation simplifiée des cas les plus courants sans nuire à ceux qui le sont moins. Précisons enfin dans ce contexte que rien n'interdit de posséder un qualifieur *constant* dont l'adéquation avec la propriété isConstant serait assurée par le méta-programmeur grâce à des assertions.

5.2.3 Atome Assertion

Une assertion est une condition qui doit être vérifiée dans un certain contexte. *OFL* offre trois genres d'assertion :

les préconditions Elles doivent être contrôlées avant l'exécution d'une primitive. Elles signalent donc les conditions nécessaires (et, si ces assertions sont complètes, les conditions sont alors nécessaires et suffisantes) pour que la primitive réalise correctement sa tâche.

les postconditions Elles participent également à la définition de la sémantique d'une primitive et doivent être vérifiées après l'exécution de celle-ci. Elle décrivent donc le travail effectué par la primitive et le résultat rendu par les accès en lecture d'un attribut ou les fonctions.

les invariants Il s'agit d'invariants de description (dits souvent *de classe*). Ils expriment des conditions qui doivent être vraies à tout moment pour chaque instance (directe ou indirecte) de la description.

Nous envisageons de créer un sous-type d'Assertion pour chacun des trois genres. Cette décomposition plus fine n'est pas ici développée car elle n'apporte rien à la compréhension du modèle. Elle est cependant particulièrement pertinente pour associer à chaque genre d'assertion une ou plusieurs politiques d'évaluation performantes[35].

Une assertion, et c'est tout son intérêt, peut toujours être considérée selon trois points de vue :

1. comme une partie significative de la documentation,
2. comme une définition formalisée du code ou
3. comme un dispositif de contrôle de la validité de l'exécution de ce code.

La structure d'Assertion est composée de :

– name : string ; Chaque assertion dispose d'un nom unique au sein de la description ou de la primitive qui la définit. Ce nom est particulièrement utile pour rendre pertinent tout message issu de la violation de l'assertion.
– condition : BooleanExpression ; Chaque assertion est décrite sous la forme d'une expression booléenne. Si le calcul de l'expression, dans son contexte d'exécution, rend true, l'assertion est vérifiée. Sinon, l'assertion est violée et une exception est émise pour le signaler. Il est envisageable de récupérer cette exception au sein même du modèle *OFL* (dans le corps des actions) pour définir un comportement spécifique en cas de problème particulier. Une assertion étant décrite par une expression booléenne, le programmeur a le choix entre :

  définir une seule *grosse* assertion En utilisant l'opérateur $\wedge$ (et), toutes les conditions peuvent être regroupées ce qui évite leur multiplication.

  définir une assertion par contrôle Chacune d'entre elles effectue donc un contrôle très spécifique ce qui facilite le pistage de l'erreur.

  faire un compromis C'est évidemment la solution la plus réaliste. Une assertion vérifie alors un comportement sémantique ; elle est selon les nécessités, composée ou atomique.

– enabled : boolean ; Selon la volonté du programmeur ou du testeur voire de l'utilisateur final, chaque assertion peut être activée (enabled est alors à true) ou désactivée (false).
– qualifiers : set of string ; Des qualifieurs peuvent être associés aux assertions. La sémantique des qualifieurs d'assertions, comme celle des autres qualifieurs, est donnée dans les concepts-descriptions ou concepts-langages.

Atome Method

Method est lui aussi un descendant de Feature. Il représente toutes les méthodes : les fonctions aussi bien que les procédures.

En plus de Feature, nous trouvons dans Method :

– parameters : set of Parameter ; Il s'agit de l'ensemble ordonné (ordre de déclaration) des paramètres formels de la méthode.
– preconditions : set of Precondition ; Chaque méthode dispose d'une précondition formé d'un ensemble de conditions. Chaque condition doit être vérifiée pour que l'exécution de la méthode se réalise dans une situation normale.
– postconditions : set of Postcondition ; C'est le pendant de la propriété précédente. La postcondition d'une méthode est l'ensemble des conditions qui doivent se vérifier si la méthode est exécutée dans le contexte d'une précondition valide.
– isConstructor : boolean ; Cette propriété signale si la méthode est un constructeur : elle vaut true dans ce cas et false sinon. Il est important de le savoir car les constructeurs disposent d'une politique d'exécution spécifique.
– isDestructor : boolean ; idem pour un destructeur.

- isOnce : boolean ; Cette propriété est à true si la méthode est à *usage unique*, c'est-à-dire que son effet n'a lieu que lors du premier appel. Si cette méthode est une fonction, les appels suivants le premier rendent l'objet calculé par le premier.
- body : set of Statement ; Le corps des méthodes est conservé sous la forme d'un ensemble ordonné d'instructions. Il est envisageable, dans un premier temps et en première approche, de conserver les instructions sous la forme d'un texte, dans la mesure où celui-ci est spécifié dans le langage cible de l'application décrite. Cependant, chaque instruction pourra par la suite, dans une évolution du modèle, être complètement réifiée.

Atome Function Function est un descendant de Method. Function décrit évidemment les fonctions (méthodes ayant un résultat au contraire des procédures). Nous y trouvons spécifiquement :

- result : UseRelationship ; Il s'agit de la relation d'utilisation qui lie la description (source) définissant la fonction à celle (cible) qui représente le type du résultat de la fonction.
- onceResult : Object ; Il s'agit simplement de la sauvegarde de l'objet calculée par la méthode si celle-ci répond true à isOnce. Lors du premier appel, cette valeur est alors initialisé ; lors des suivants, elle est utilisée.

Atome Procedure Comme Function, Procedure hérite de Method. Mais contrairement à Function, il n'est rien nécessaire d'ajouter à Method pour décrire une procédure. Nous suggérons cependant, par l'existence de l'atome Procedure, que l'atome Method conserve, comme Feature, un aspect abstrait et que chacune des ses primitives soit donc instance directe d'Attribute, Function ou Procedure.

5.2.4 Atome Parameter

L'atome Parameter réifie la notion de paramètre formel d'une primitive.
Un Parameter est constitué des éléments suivants :

- name : string ; Il s'agit du nom du paramètre formel.
- relation : UseRelationship ; Chaque paramètre formel décrit une relation d'utilisation entre la description (source) qui définit la méthode et la description (cible) représentant le type du paramètre.
- isInput : boolean ; Cette propriété booléenne est à true si le paramètre courant est une *donnée* pour la méthode, c'est-à-dire si sa valeur initiale est prise en compte. Il est à false sinon.
- isOutput : boolean ; isOutput est à vrai si et seulement si le paramètre courant est un résultat pour la méthode, c'est-à-dire si sa valeur à la sortie de la méthode est significative. Plusieurs contrôles peuvent découler de la valeur de ces deux dernières propriétés :
  - Si le paramètre est une donnée mais pas un résultat, nous pouvons envisager d'obliger le paramètre à être initialisé avant d'être passé à la méthode, empêcher la modification de sa valeur, ...
  - Si le paramètre est un résultat mais pas une donnée, il est par exemple possible de vérifier qu'il est initialisé avant la sortie de la méthode.
  - Si le paramètre est une donnée et un résultat, d'autres vérifications peuvent être faites.

  - Enfin, si isInput et isOutput sont à false, cela signifie que la sémantique d'utilisation des paramètres n'est pas donnée par le programmeur mais par le langage de programmation (comme c'est souvent le cas).
- qualifiers : set of string ; De la même manière que pour les primitives, il est possible d'associer des qualifieurs particuliers aux paramètres formels de celles-ci. isInput et isOutput auraient pu être gérés par qualifiers mais nous avons choisi, en raison de leur importance, de le faire à part. Comme ceux des primitives, la sémantique des qualifieurs des paramètres est exprimée par le méta-programmeur au moyen d'assertions au sein des concepts-descriptions ou concepts-langages.

Les paramètres effectifs, qui sont des entités présentes à l'exécution de la primitive, sont modélisés plus simplement comme des objets de l'application. Leur type, accessible par l'intermédiaire de la relation d'instanciation, permet notamment de réaliser la traditionnelle vérification de conformité entre paramètres effectifs et formels.

5.2.5 Atome Message

Un message est une *demande de travail* envoyé par un objet à un autre au moyen d'une instruction[8]. Certains messages, les fonctions ou les accès en lecture à un attribut, requièrent un résultat. D'autres, les procédures ou les accès en écriture à un attribut, n'en ont pas besoin. Un message peut être structuré de la manière suivante :

- target : Object ; Il s'agit du destinataire du message. Par exemple, dans une expression telle un.deux().trois, il s'agit de un.
- callExpression : Expression ; Il s'agit de l'expression proprement dite qui, appliquée à target, constitue la demande de travail. Dans un.deux().trois, cette expression est deux().trois. Cette expression peut elle-même, et c'est le cas dans notre exemple, représenter un message qui doit alors être évalué préalablement (l'associativité de l'opérateur d'envoi de message . étant de droite à gauche).

Remarquons qu'au premier niveau l'instruction un.deux().trois s'applique avec target égal à l'objet courant et callExpression égal à toute l'instruction. En effet, celle-ci est équivalente à currentObject[9].un.deux().trois.

5.2.6 Atome Type

Chaque description représente un ou plusieurs types : si elle est générique, elle en représente plusieurs, sinon un seul. Les relations que nous qualifions généralement d'*inter-descriptions* sont ainsi dans les faits parfois des relations *inter-types*. Un type est décrit notamment par :

- description : Description ; Chaque type est modélisé par une description.
- isGeneric : boolean ; Cette propriété est à true si description est générique, à false sinon. Rappelons que la capacité à être générique pour une description est conditionnée par un paramètre de concept-description. Il est possible de se passer de isGeneric dans Type, cette information étant également disponible dans description.

[8] Comme nous l'avons déjà dit, la réification des instructions, des expressions, ... n'a pas besoin d'être complète dans l'état actuel du modèle, nous ne la décrivons donc pas dans cette section.

[9] currentObject représente l'objet courant.

- genericParameters : set of EffectiveGenericParameters ; Si isGeneric est à true, EffectiveGenericParameters contient la liste ordonnée de ses paramètres génériques effectifs.
- directExtension : set of Object ; directExtension est l'ensemble des instances directes du type courant.
- fullExtension : set of Object ; Il s'agit de l'extension complète du type courant, c'est-à-dire de l'union de directExtension et des extensions complètes de tous ses sous-types. Sa valeur dépend donc des relations d'importation mise en œuvre autour de la description qui représente ce type.

5.2.7 Atome Language

L'atome Language est la réification d'un langage de programmation. Il s'agit d'une entité abstraite, les composants du langage lui-même étant principalement ses descriptions et ses relations.

Un langage possède les propriétés spécifiques suivantes :

- name : string ; Pour des raisons évidentes d'introspection, chaque langage est nommé.
- syntax : Grammar ; Une grammaire spécifiant la syntaxe du langage peut être associée à cet atome. Il est ainsi possible d'utiliser plusieurs outils (compilateurs, interprètes, ...) génériques pour traduire ce langage. Cette information n'est pour l'heure pas exploitée mais utilisable comme documentation et prévue pour de futures améliorations. Le format de la syntaxe peut encore évoluer, par exemple vers une modélisation *décorée* permettant de spécifier certains éléments sémantiques liés à la syntaxe.

5.2.8 Atome Description

Description est le deuxième atome, après Language, est la réification d'une entité dont il existe un concept (ici, il s'agit du concept-description.

Une description possède la structure suivante au sein d'*OFL* :

- name : string ; Il s'agit simplement du nom de la description.
- sourceRelationships : set of Relationship ; Chaque description a connaissance de l'ensemble des relations pour lesquelles elle est utilisée en tant que source. Cet ensemble est sourceRelationships.
- targetRelationships : set of Relationship ; Idem que sourceRelationships mais pour les relations qui utilisent la description courante en tant que cible.
- features : set of Feature ; Il s'agit de la liste des primitives (attributs et méthodes) propres de la description.
- isGeneric : boolean ; Ce booléen est à true si la description courante est générique (i. e. possède au moins un paramètre de généricité), à false sinon.
- formalGenericTypes : set of FormalGenericParameters ; Si isGeneric est à true, formalGenericTypes contient la liste ordonnée de ses paramètres génériques formels. Ceux-ci sont à mettre en adéquation avec les paramètres génériques effectifs présents dans chaque membre de effectiveTypes.
- effectiveTypes : set of Type ; effectiveTypes représente l'ensemble des types effectifs décrits par la description courante. Si isGeneric (cf. ci-dessous) est à false, cet ensemble est un monôme ; sinon, plusieurs types sont possibles en fonction de l'utilisation des paramètres génériques (cf. formalGenericTypes ci-dessus) de la description.

- directExtension : set of Object ; directExtension est l'ensemble des instances directes de la description courante, c'est-à-dire l'union des extensions de ses types.
- fullExtension : set of Object ; Il s'agit de l'extension complète de la description courante, c'est-à-dire de l'union de directExtension et des extensions complètes de toutes ses sous-descriptions (c'est donc aussi l'union des extensions des types de la description courante et de toutes ses sous-descriptions).
- invariants : set of Invariant ; Il s'agit d'un ensemble de conditions qui doivent être vérifiées à tout moment pour toutes les instances (directes et indirectes) de la description.
- qualifiers : set of string ; Comme pour les primitives et les paramètres, des qualifieurs peuvent être associés aux descriptions. qualifiers est l'ensemble des qualifieurs de la description courante. La liste des qualifieurs possibles est donnée dans le concept-description ou le concept-langage.

5.2.9 Atome Relationship

Troisième et dernier atome possédant son pendant dans les concepts, Relationship réifie les relations (principalement entre descriptions) au sein d'une application décrite par un langage à objets.

Relationship est structuré ainsi :

- name : string ; C'est le nom de la relation.
- sourceDescriptions : set of Type ; Chaque relation possède un ensemble décrivant la liste des descriptions (et, plus précisément des types) qui lui servent de sources.
- targetDescriptions : set of Type ; Pendant de la propriété précédente, il s'agit de l'ensemble des types qui sont cibles de la relation courante.
- removedFeatures : set of Feature ; Une relation lie des sources à des cibles. Les primitives des cibles sont importées ou utilisées par les sources. Cette importation ou cette utilisation peut être *adaptée* par la relation ; ainsi des primitives peuvent être supprimées (une primitive supprimée est donc présente dans une cible mais pas dans une source) par l'importation ou l'utilisation. L'ensemble des primitives supprimées est removedFeatures. Rappelons que la suppression de primitives n'est généralement pas possible dans les langages traditionnels.
- renamedFeatures : set of <Feature, string> ; Sur le même modèle que removedFeatures, l'ensemble des primitives renommées par la relation courante est conservé dans renamedFeatures, associé à leur nouveau nom.
- Avec la même idée que removedFeatures et renamedFeatures, nous retrouvons dans cette réification de relation :
  - addedFeatures,
  - redefinedFeatures,
  - hiddenFeatures,
  - shownFeatures,
  - abstractedFeatures et
  - effectedFeatures.
- qualifiers : set of string ; Les relations aussi peuvent être qualifiées.

L'atome Relationship possède naturellement plusieurs descendants. Pour vous remettre cela en mémoire, revenez à la figure 5.2 page 107.

Chapitre 6

Actions

Les actions (cf. définition 14 page 75), dans le modèle *OFL*, constituent l'ensemble des algorithmes de base permettant de mettre en œuvre les paramètres. Chaque action réalise en effet une tâche bien déterminée en fonction de la valeur d'un ou plusieurs paramètres ayant une incidence sur cette tâche.

Les actions sont attachées aux concepts-descriptions et concepts-relations. Le concept-description étant, pour nous, une entité de plus haut niveau que le concept-relation, chacune de ses actions est généralement une composition des actions homonymes du concept-description. Nous rappelons que nous avons classé les actions par thème. Ce classement est arbitraire et n'est donné que pour structurer et simplifier la présentation.

6.1 Protocole de méta-programmation

Nous proposons, lors de l'implémentation des actions, un protocole aussi systématique que possible, instauré dans le but de faciliter leur usage et leur manipulation. Toute action *AC* (exemples que nous spécifierons plus loin : lookup, assign) est donc définie par les éléments suivants.

- Une primitive booléenne is_ac_valid donne true si *AC* possède une sémantique dans le concept-langage courant, false sinon.
- Une primitive ac décrit l'implantation de la sémantique associée à *AC*.
- Pour les actions de concept-relation seulement : un ensemble de primitives ac_l (où l est une lettre) qui définit des étapes de sous-comportement (cf. section 6.2 page suivante) pour ac.
- Un ensemble de primitives default_ac_x (où x est un entier naturel) qui fournit des comportements standards pour ac, default_ac_0 décrivant généralement le comportement par défaut de ac. Ce système de comportement standard est extensible aux sous-comportements des actions de concept-relation.

La présence de comportements standards pour chacune des actions[1] doit permettre de réduire au maximum les interventions du méta-programmeur sur les actions. Il lui suffit en effet, dans tous les cas déjà prévus par le système, de sélectionner les comportements qu'il veut voir appliquer. Le protocole que nous venons de présenter peut néanmoins aussi servir de cadre lorsque l'ajout d'actions s'avère

[1]Nous entendons par « comportements standards d'une action », l'ensemble des algorithmes qui se retrouvent généralement implantés dans les langages de programmation courants pour réaliser cette action.

indispensable. Mais *OFL* étant défini pour couvrir un large champ de langages de programmation, cela doit rester rare. Ce protocole de méta-programmation s'ajoute aux paramètres hyper-génériques définis précédemment pour rendre le travail du méta-programmeur plus simple et plus efficace.

6.2 Facettes statiques et dynamiques

Une action est principalement un algorithme. Cette définition est simple mais cache une implémentation qui ne l'est pas forcément. En effet, nous venons de voir la mise en place d'un protocole permettant de décrire plusieurs *versions* de la même action. Mais ce n'est pas tout. La réalisation d'une action spécifique comporte en fait deux facettes.

une facette statique Il s'agit de la partie qu'il est possible de réaliser une fois pour toute lors d'une compilation par exemple. Pour illustrer ce fait dans le cadre de la liaison dynamique, nous pouvons décider de placer dans cette facette la partie calcul du graphe de types, voire pré-calculer tous les appels ne laissant pas d'ambiguïté sur le type de l'objet receveur. La facette statique est généralement dévolue à un compilateur.

une facette dynamique Au contraire, d'autres parties de l'application ne peuvent être réalisées que lors de l'exécution proprement dite. Dans un langage à objets, le choix de la *bonne* primitive à activer est fréquemment repoussée au moment de l'exécution (c'est pour cela que nous parlons de liaison *dynamique*). Cette partie-là de l'action sera donc intégrée à la facette dynamique. La facette dynamique est souvent laissée à la charge d'un interprète ou d'une machine virtuelle.

Précisons aussi que les interactions entre facettes dynamiques et statiques d'une action peuvent être implémentées de diverses façons. Dans l'état actuel de notre étude, un choix strict n'a pas encore été réalisé en ce domaine. En tout état de cause, et quelque soit le choix effectué, la répartition des tâches entre facettes statiques et dynamiques est en grande partie arbitraire et donc à la charge du méta-programmeur responsable de l'implémentation des actions.

En plus de cette décomposition en deux facettes, ajoutons que les actions de concept-description font généralement appel à leur équivalent dans les concepts-relations. Par exemple, l'action d'envoi de message des concepts-descriptions peut être constituée d'une composition plus ou moins élaborée des actions d'envoi de message des concepts-relations. Mais, pour ces dernières, la granularité doit être plus fine : il faudra par exemple réaliser certaines tâches spécifiques aux relations, puis exécuter le cœur de l'action de concept-description puis réaliser d'autres opérations spécifiques aux relations. À cet effet, nous décrivons ces tâches par un ensemble de *sous-comportements* fédérés par une action qui les appelle au moment propice. Nous vous proposons de vous référer à la figure 6.1 page suivante[2] pour y découvrir un exemple.

6.3 Avant, après et autour

Le modèle *OFL* est amené à évoluer, à s'adjoindre de nouvelles capacités. Ainsi, la thèse de doctorat actuellement préparée par Adeline Capouillez [10] envisage l'inté-

[2]Rappelons qu'une légende globale des figures est donnée dans la figure 3.1 page 48.

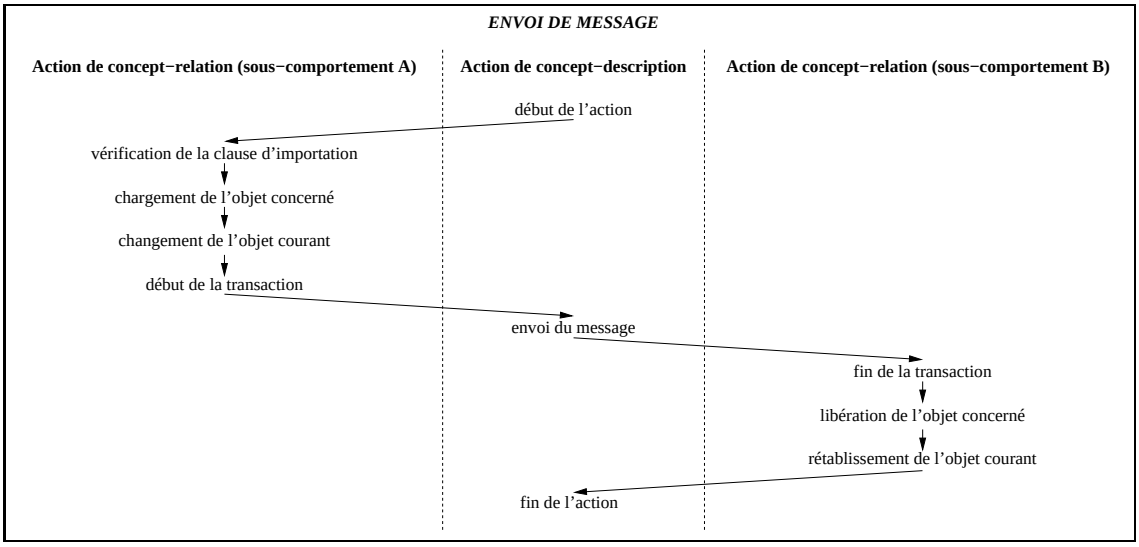

FIG. 6.1 – Exemple d'une action de concept-relation

gration des principes de la programmation par aspects au sein du modèle. Ce travail, ainsi que la volonté de conserver le modèle le plus ouvert possible, nous ont amené à permettre la création d'un mécanisme basé sur des before, after et around tels ceux de *CLOS* pour les actions.

6.3.1 before

Un before est un ensemble d'instructions accolé à une action et qui doit être exécuté avant elle. Un before ne modifie pas le code d'une action, il le fait précéder d'une partie systématique. Il est possible d'ajouter autant de befores que nécessaire à une action, ceux-ci étant alors exécutés dans l'ordre inverse de leur ajout (le dernier ajouté est exécuté en premier).

6.3.2 after

Un after est l'équivalent d'un before mais se place après et non avant l'action. Si plusieurs afters sont déclarés, leur exécution a lieu dans leur ordre d'ajout (le dernier ajouté est exécuté en dernier).

6.3.3 around

Un around est une encapsulation du code d'une action. En effet, certains ajouts ne peuvent être limités à une action avant et/ou après le code modifié. Le pseudo-code suivant donne un exemple simpliste d'around : **si** <code_actuel> == 3 **alors** déclencher(exception3) **finsi**. Les arounds peuvent être ajoutés les uns autour des autres, l'exécution du dernier se faisant autour de celle de l'avant-dernier et ainsi de suite.

6.3.4 Précédence

Tout ajout de before, after ou around peut (et souvent doit) naturellement s'accompagner d'une spécification plus précise de l'action et donc d'un relâchement de sa

précondition et d'un renforcement de sa postcondition.

Enfin, l'ordre global d'exécution est celui-ci :

1. exécution des éventuels befores dans l'ordre inverse de leur ajout, puis
2. exécution des éventuels arounds dans l'ordre inverse de leur ajout (et ce jusqu'au code initial encapsulé) ou exécution du code initial si aucun around n'est présent, puis
3. exécution des éventuels afters dans l'ordre de leur ajout.

Dans tous les cas, les modifications apportées par ce mécanisme doivent rester transparentes pour l'utilisateur des actions, qui se trouve être le programmeur de l'application finale. Cette condition est la raison qui fait de ces trois qualifieurs de simples ajouts de code et en aucun cas une modification de la signature de l'action.

Voyons maintenant la liste des actions du modèle *OFL*.

6.4 Actions de recherche de primitive

La première catégorie d'actions présentées ici concerne la recherche de primitive. Ces actions sont les suivantes.

match

définition match vérifie la correspondance (nom, genre [constructeur, destructeur, attribut, fonction, procédure, ...] et paramètres) entre une primitive déclarée et un appel à une primitive.

signature boolean match(F : Feature, M : Message)
F est la primitive déclarée et M le message d'un appel à une primitive. Le résultat est true si F correspond à M, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) are_valid_parameters (cf. page 134), de are_valid_generic_parameters (cf. page 135), de is_valid_generation (cf. page 137) et de is_valid_destruction (cf. page 137).

relation (cf. pages 85–91) ceux de are_valid_parameters (cf. page 134), de are_valid_generic_parameters (cf. page 135), de is_valid_generation (cf. page 137) et de is_valid_destruction (cf. page 137).

remarques

– match fait généralement appel à are_valid_parameters (cf. page 134) pour contrôler l'adéquation des paramètres entre F et M lorsque il s'agit de vérifier une routine. Pour un attribut, si celui-ci est générique, il peut être fait appel à are_valid_generic_parameters (cf. page 135). is_valid_generation (cf. page 137) est également utilisée dans le cas où la primitive appelée est un constructeur, pour vérifier que cet appel est valide en fonction de la sémantique du concept-description. Il est en effet tout à fait possible de définir des descriptions possédant des constructeurs qui ne peuvent être appelés directement, comme par exemple les constructeurs des classes abstraites en *Java* (cf. section 3.3.3 page 66). Pour une raison similaire concernant les destructeurs, is_valid_destruction (cf. page 137) est appelée.

- Il est important de noter que le fait que match rende true ne signifie pas que la primitive F soit celle qu'il faut appliquer à M. F correspond certes à M mais d'autres primitives peuvent aussi correspondre et être mieux adaptées. C'est l'action local_lookup (cf. page 123) qui sélectionne parmi toutes les primitives candidates (pour lesquelles match a rendu true) celle qui sera appliquée.
- Par défaut, le polymorphisme ne s'applique pas aux constructeurs et aux destructeurs. Mais cela pourrait être adapté par la redéfinition de match voire par l'ajout d'un paramètre de concept-relation.

local_features

définition local_features permet de déterminer l'ensemble des primitives accessibles localement (sans tenir compte des relations entre descriptions) aux instances d'une description.

signature set of Feature local_features()
Le résultat est l'ensemble des primitives accessibles localement aux instances d'une description.

paramètres pris en compte dans une

description (cf. pages 79–81)
- Encapsulation pour rechercher dans la description directement ou dans une structure externe et
- Overloading pour prendre en compte l'éventuelle surcharge entre les primitives de la description.

relation (cf. pages 85–91) aucun.

remarques
- Aucune relation n'est prise en compte : ni utilisation, ni importation. Le paramètre de concept-relation Indirect_access n'est donc pas utilisé ici.
- L'usage du paramètre de concept-description Visibility et des paramètres de concept-relation Direct_access, Read_accessor et Write_accessor n'est pas envisagé car nous considérons que toutes les primitives d'une description sont forcément accessibles aux instances de celle-ci.

all_features

définition all_features permet de déterminer l'ensemble des primitives accessibles aux instances d'une description en tenant compte des relations entre descriptions.

signature set of Feature all_features()
Le résultat est l'ensemble des primitives accessibles aux instances d'une description.

paramètres pris en compte dans une

description (cf. pages 79–81) en plus de ceux de local_features (cf. de la présente page)
- Visibility pour déterminer la visibilité des descriptions des primitives par les instances de la description courante (la visibilité de la primitives elle-même, au sein de sa description, est gérée par un attribut de l'objet représentant cette primitive, pas par un paramètre hypergénérique) et

  - Overloading pour prendre en compte l'éventuelle surcharge entre les primitives.

relation (cf. pages 85–91) en plus de ceux de local_features (cf. page précédente)
  - Kind pour suivre les relations d'importation (et les quelques relations d'utilisation pour lesquelles Direct_Access n'est pas égal à forbidden),
  - Cardinality pour parcourir plusieurs branches des graphes en cas de relation multiple,
  - Circularity pour éviter d'entrer dans des boucles infinies dans les relations circulaires,
  - Opposite pour éventuellement suivre les relations opposées de la cible vers la source,
  - Direct_access et Indirect_access pour vérifier si les primitives sont accessibles et sous quels noms,
  - Polymorphism_implication et Polymorphism_policy pour déterminer les relations à suivre, dans quel sens et selon quelle politique,
  - Feature_variance pour éviter des recherches que la politique de variance rend forcément infructueuses,
  - Read_accessor et Write_accessor lorsque la primitive rencontrée est un attribut, pour déterminer s'il est accessible et
  - Adding, Removing, Renaming, Redefining, Masking, Showing, Abstracting et Effecting pour tenir compte de la manière dont la primitive est importée (ou utilisée).

remarques
- all_features fait généralement appel à local_features (cf. page précédente) pour débuter par une recherche locale dans la description courante. Viennent ensuite se greffer les primitives issues des relations d'importation ainsi que de certaines relations d'utilisation (pour lesquelles le paramètre Direct_access n'est pas égal à forbidden).
- all_features est une méthode qui peut s'avérer lourde. Il lui faut en effet parcourir tous les graphes de tous les concepts-relations en partant de la description courante. Cependant, le nombre réel d'importations et d'utilisations est souvent limité dans une description par le simple fait que celle-ci doit rester compréhensible par les programmeurs qui l'utiliseront.
- De nombreuses optimisations peuvent être envisagées qui réduisent notablement le nombre de fois où all_features calcule cet ensemble de primitives : pré-parcours, conservation des calculs déjà effectués, des calculs les plus fréquents ou les plus longs, . . . Bon nombre d'entre elles trouvent leur place dans la facette statique de l'action.
- Nous proposons de réaliser dans la facette statique de all_features, les actions de contrôle suivantes. Mais attention : lancer toutes ces vérifications à chaque fois n'aurait aucun sens ! Il serait plus cohérent de les lancer seulement lors dur premier appel à all_features par exemple.
  - verify_genericity (cf. page 131) si la description est générique, pour vérifier qu'elle est en droit de l'être.
  - verify_cardinality (cf. page 133) pour contrôler que la cardinalité du concept-relation est bien respectée par les relations parcourues.

- verify_circularity (cf. page 133) pour vérifier que les relations parcourues sont circulaires ou non en fonction de la sémantique du concept-relation concerné.
- verify_repetition (cf. page 133) pour s'assurer que la répétition dans les descriptions-sources et les descriptions-cibles des relations parcourues est conforme à la valeur du paramètre Repetition.
- verify_variance (cf. page 134) pour vérifier que la variance des différentes composantes des primitives (paramètres, résultats de fonction et attributs) et des assertions (invariants, préconditions et postconditions) est conforme avec la sémantique du concept-relation sur ce point.
- verify_adaptation (cf. page 134) permet de contrôler que les adaptations de primitives (suppression, renommage, redéfinition, masquage, démasquage, abstraction ou encore concrétisation) sont mises en œuvre conformément aux paramètres de concept-relation correspondants.

local_lookup

définition local_lookup recherche localement (indépendamment de toute relation d'importation) dans la description courante, la primitive à appliquer suite à un message.

signature Feature local_lookup(M : Message)
M est le message de l'appel à une primitive. Le résultat est la primitive locale sélectionnée ou la référence nulle (que nous appellerons désormais null) si aucune primitive ne convient.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de local_features (cf. page 121) et match (cf. page 120).

relation (cf. pages 79–81) en plus de ceux de local_features (cf. page 121) et match (cf. page 120)
- Polymorphism_implication pour déterminer un graphe de types en fonction des relations d'importation polymorphiques. En effet, même si local_lookup réalise une recherche locale (à une description), il est utile de connaître le graphe de types pour contrôler la validité du type des paramètres effectifs par rapport à celui des paramètres formels.

remarques
- local_lookup fait généralement appel à local_features (cf. page 121) pour déterminer l'ensemble des primitives accessibles localement puis à match (cf. page 120) pour contrôler l'adéquation entre chaque primitive accessible localement et le message M.
- local_lookup sélectionne parmi les primitives candidates (celles pour lesquelles match a rendu true) celle qui est la plus adaptée en fonction du graphe de types (déduit du graphe de polymorphismes). Cet algorithme de sélection peut être complexe et parfois mettre en avant des choix arbitraires, par exemple en utilisant des listes de précédences, pour déterminer le meilleur de plusieurs possibilités *a priori* équivalentes.

– Pour la détermination du graphe de types (pour vérifier l'adéquation du types des paramètres effectifs par rapport à celui des paramètres formels), nous proposons de suivre les relations polymorphiques. Si A est source d'une relation R dont B est cible et que le paramètre Polymorphism_implication de R indique up alors A est un sous-type de B. Si Polymorphism_implication est à down, alors B est un sous-type de A. Enfin, si la valeur de ce paramètre est both alors A et B peuvent être considérés comme équivalents du point de vue des types.
– Enfin, précisons que l'action local_lookup est d'un usage plus marginal que lookup (cf. de la présente page), car elle *désactive* la liaison dynamique (mais pas la surcharge). Elle peut cependant être très utile en phase d'expérimentation d'un concept-langage ou pour simuler des langages sans polymorphisme. local_lookup peut aussi servir dans lookup, pour effectuer une première recherche performante avant de se lancer dans une recherche complète mais lente.

lookup

définition lookup recherche la primitive correspondant à un message en tenant compte des relations inter-descriptions.

signature Feature lookup(M : Message)
M est le message donnant la signature de la primitive à rechercher. Le résultat est la primitive sélectionnée ou null si aucune primitive ne convient.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de all_features (cf. page 121) et match (cf. page 120).

relation (cf. pages 85–91) en plus de ceux de all_features (cf. page 121) et match (cf. page 120)
– Polymorphism_implication pour déterminer un graphe de types en fonction des relations d'importation polymorphiques.

remarques
– lookup fait généralement appel à all_features (cf. page 121) pour déterminer l'ensemble des primitives accessibles puis à match (cf. page 120) pour contrôler l'adéquation entre chaque primitive accessible et le message M.
– L'action lookup fait ensuite un travail similaire à local_lookup (cf. page précédente) : elle sélectionne la primitive la plus adaptée pour répondre au message M. La seule différence est que cette sélection s'applique ici en tenant compte des relations inter-descriptions.
– Le calcul du graphe de types (pour vérifier l'adéquation du types des paramètres effectifs par rapport à celui des paramètres formels) dans lookup peut être totalement identique à celui de local_lookup (cf. page précédente). Si c'est le cas, il peut être dévolu à une routine auxiliaire.
– Un exemple de pseudo-code de l'action de concept-description lookup est donné section 6.10 page 147.
– L'appel des actions de contrôle suivantes est possible. Cependant, il ne s'agit pas de les utiliser à chaque exécution de lookup, bien au contraire. En effet toutes ses vérifications sont globales (elles s'appliquent à toutes

les primitives et non seulement à une seule) et doivent donc être effectuées avec parcimonie. D'autant plus que, dans une application structurée statiquement (sans modification des descriptions à l'exécution), le résultat de ces vérifications ne varie jamais. Précisons aussi que toutes ces vérifications peuvent également être réalisées par un compilateur plutôt qu'à l'exécution, et donc dans la facette statique de l'action lookup. Nous suggérons de faire ici (dans lookup) ces contrôles car ils concernent les appels à une primitive mais ils peuvent tout aussi bien être réalisés ailleurs.

- verify_generator (cf. page 131) pour vérifier qu'un appel au constructeur est valide.
- verify_destructor (cf. page 132) à cause d'une raison similaire concernant les destructeurs.
- verify_encapsulation (cf. page 132) qui contrôle que la politique d'encapsulation des primitives par la description est bien respectée.
- verify_attribute (cf. page 132) pour contrôler que l'éventuelle présence d'attribut est conforme à la sémantique de la description.
- verify_method (cf. page 132) pour une raison similaire concernant les méthodes.
- verify_overloading (cf. page 132) pour vérifier que l'usage de la surcharge est compatible avec la sémantique de la description.

Notons qu'il aurait été envisageable de définir un paramètre pour décrire certains choix réalisés (par exemple, la possibilité de recourir à une liste de précédences pour finalement sélectionner la primitive adéquate). C'est presque toujours vrai lorsque nous nous trouvons en face de plusieurs alternatives d'implantation d'une action. Cependant, comme il aurait été déraisonnable d'augmenter inconsidérément le nombre de paramètres, nous avons choisi, lorsque les alternatives nous paraissent d'une importance moindre, de les traiter sous la forme d'un ensemble de comportements standards pour les actions. Notre découpage est donc raisonné mais arbitraire et pourra être remis en question si l'expérience nous en dévoile un meilleur.

6.5 Actions d'exécution de primitive

Les actions d'exécution de primitive assurent la gestion des échanges de message entre objets et de l'exécution des instructions. Avant toute exécution au travers de ces actions, les actions de recherche de primitive (section 6.4 page 120) sont évidemment utilisées. Nous faisons aussi un usage notable des actions de contrôle (section 6.6 page 130).

evaluate_parameters

définition evaluate_parameters évalue les paramètres associée à une primitive avant l'exécution d'une primitive.

signature set of object evaluate_parameters(SE : set of Expression, M : Message)
SE est l'ensemble ordonné des expressions décrivant les paramètres à évaluer par la primitive décrite dans le message M. Le résultat est un ensemble, dans le même ordre que SE, du résultat de l'évaluation de chaque expression.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de submit_with_result (cf. page 130).

relation (cf. pages 85–91) ceux de submit_with_result (cf. page 130).

remarques
- evaluate_parameters fait généralement appel à submit_with_result (cf. page 130) pour évaluer chacun des paramètres de méthode.
- evaluate_parameters est notamment utile pour gérer un ordre particulier d'évaluation des paramètres de méthode. Ce choix pourra d'ailleurs faire l'objet d'un paramètre de concept-description (paramètre hyper-générique alors utilisé par evaluate_parameters) si cela s'avère utile.

attach_parameters

définition attach_parameters réalise l'attachement des paramètres effectifs au début de l'exécution d'une primitive.

signature void attach_parameters(SO : set of object, M : Message)
SO est l'ensemble ordonné des paramètres évalués et M est le message décrivant la primitive exécutée. attach_parameters réalise l'attachement des objets présents dans SO (paramètres effectifs) aux paramètres (jusque là formels) de la primitive de M. attach_parameters ne rend rien (cela est signalé par void).

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91)
- Dependence pour adapter la méthode d'attachement des paramètres à la sémantique de la relation d'utilisation mise en œuvre.

remarques – Nous envisageons éventuellement, pour améliorer encore la précision du système de paramètres hyper-génériques, d'ajouter un paramètre de concept-description indiquant si les méthodes doivent effectuer par défaut un attachement de paramètres par valeur (comme ici) ou par référence.
- attach_parameters doit également prendre en considération les éventuels qualifieurs de paramètres (tels const, in, out, . . .) qui peuvent apparaître dans le langage modélisé. Mais le comportement de ces qualifieurs n'étant pas paramétrable par *OFL*, leur sémantique opérationnelle est laissée à la charge du méta-programmeur.

detach_parameters Cette action réalise l'opération inverse de attach_parameters (cf. de la présente page), elle détache les paramètres effectifs. Pour respecter la sémantique mise en œuvre par attach_parameters, elle utilise également le paramètre de concept-description Dependence.

before_execute

définition before_execute réalise toutes les tâches utiles avant l'exécution proprement dite d'une primitive.

signature void before_execute(M : Message)
before_execute est, entre autres, chargé de réaliser, avant exécution de la primitive décrite dans M, les tâches suivantes :
– contrôle de la précondition de la primitive et de l'invariant de sa description (s'il a été décidé de l'évaluer avant toute exécution de primitive),
– remplacement de l'objet courant par celui sur lequel est exécutée la primitive,
– évaluation et attachement des paramètres effectifs,
– déclenchement d'une transaction éventuelle (par exemple si l'exécution est faite dans un contexte de langage ayant activé la classe de services de persistance),
– ...

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et attach_parameters (cf. page ci-contre).

relation (cf. pages 85–91) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et attach_parameters (cf. page précédente).

remarques
– before_execute fait appel à check_assertion (cf. page 130) pour évaluer chaque assertion de la précondition de la primitive. Il est également fait appel à evaluate_parameters (cf. page 125) puis à attach_parameters (cf. page ci-contre) pour évaluer puis attacher les paramètres effectifs.
– Dans le cadre d'une application permettant la gestion d'objets persistants, il serait utile de prendre en compte le paramètre de concept-relation Transaction_policy (cf. page 91) pour contrôler la politique d'activation des transactions.

after_execute

définition after_execute réalise toutes les tâches utiles après l'exécution proprement dite d'une primitive. C'est le pendant de before_execute (cf. page ci-contre).

signature void after_execute(M : Message)
after_execute est, entre autres, chargé de réaliser, après exécution de la primitive décrite dans M, les tâches suivantes :
– remplacement, comme objet courant, de l'objet sur lequel a été exécutée la primitive, par l'objet qui était courant avant l'exécution,
– détachement des paramètres effectifs,
– fin d'une transaction éventuelle, si la classe de service de persistance est activée,
– contrôle de la postcondition de la primitive et de l'invariant de description (s'il a été décidé de l'évaluer après toute exécution de primitive),
– ...

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130) et detach_parameters (cf. page ci-contre).

relation (cf. pages 85–91) ceux de check_assertion (cf. page 130) et detach_parameters (cf. page 126).

remarques

- after_execute fait généralement appel à check_assertion (cf. page 130) pour évaluer chaque assertion de la postcondition de la primitive et de l'invariant de description (s'il y a lieu). Il est également fait appel à detach_parameters (cf. page 126) pour détacher les paramètres effectifs.
- De la même manière que dans before_execute (cf. page 126), l'usage du paramètre de concept-relation Transaction_policy peut s'avérer pertinent. Cet exemple de fin de transaction n'est donné qu'à titre indicatif pour permettre au lecteur d'imaginer les adaptations nécessaire à l'intégration d'une telle classe de service[3].

execute

définition execute exécute une primitive. La primitive exécutée doit être une procédure et non un attribut ou une fonction car execute ne gère pas de résultat.

signature void execute(M : Message)
execute exécute la procédure décrite dans M. Nous pouvons résumer les tâches à exécuter :

- recherche de la primitive à exécuter dans l'arbre de types,
- activation des tâches utiles avant l'exécution de la procédure,
- exécution proprement dite de la procédure,
- activation des tâches utiles après l'exécution de la procédure,

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de lookup (cf. page 124), before_execute (cf. page 126) et after_execute (cf. page précédente).

relation (cf. pages 85–91) ceux de lookup (cf. page 124), before_execute (cf. page 126) et after_execute (cf. page précédente).

remarques

- execute fait généralement appel à lookup (cf. page 124) pour rechercher la primitive adéquate à exécuter. Il s'agit, après cela, de réaliser toutes les tâches utiles à l'exécution de la procédure, cette fonction étant dévolue à before_execute (cf. page 126). Puis, l'exécution proprement dite est effectuée. Le tout se termine pas l'activation des tâches à effectuer après l'exécution d'une primitive : after_execute (cf. page précédente).
- L'exécution à proprement parler de la procédure dépend de l'intégration du modèle *OFL* au langage cible choisi (par exemple : *Java*). Si nous considérons *OFL* comme la plate-forme d'exécution, il faut alors réifier les instructions et les exécuter. Si nous nous appuyons sur un compilateur/interprète existant de *Java*, nous pouvons alors lui *déléguer* la représentation et l'exécution des instructions.

[3] Un autre thèse de doctorat, visant à intégrer ces classes de service en utilisant la *programmation par aspects*, est en cours de rédaction par Adeline Capouillez.

execute_with_result C'est le pendant de execute (cf. page précédente) pour les primitives ayant un résultat : les fonctions et non plus les procédures. La signature de execute_with_result est donc object execute_with_result(M : message).

get_attribute_value C'est lui aussi un équivalent de execute (cf. page ci-contre) et de execute_with_result (cf. de la présente page). Cependant, cette action est spécifiquement appelée pour accéder à la valeur d'un attribut. Sa signature est object get_attribute_value(M : Message).

send

définition send gère toutes les actions spécifiques à l'envoi de message : évaluation du destinataire du message, éventuelle prise en compte de la nature de l'échange (synchrone ou asynchrone), gestion d'éventuels destinataires intermédiaires (lorsqu'un message contient lui-même un ou plusieurs messages), ...

signature void send(M : Message)
send calcule le destinataire du message M et lui applique la primitive décrite dans M. Cette primitive doit être une procédure car send ne prend pas en compte de résultat.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de submit_with_result (cf. page suivante) et execute (cf. page précédente).

relation (cf. pages 85–91) ceux de submit_with_result (cf. page suivante) et execute (cf. page précédente).

remarque send fait généralement appel à submit_with_result (cf. page suivante) pour évaluer l'objet sur lequel doit être appliqué la procédure et à submit (cf. de la présente page) pour exécuter la procédure.

send_with_result C'est l'équivalent de send (cf. de la présente page) pour les primitives ayant un résultat (attributs ou fonctions mais pas procédures). Sa signature est object send_with_result(M : Message) et submit_with_result (cf. page suivante) (ou get_attribute_value [cf. de la présente page]) est utilisé plutôt que submit.

submit

définition submit soumet un message à un objet. submit ne peut prendre en compte un message que si celui-ci décrit une procédure car submit ne gère pas de résultat.

signature void submit(M : Message)
submit a pour charge de répartir le travail entre :
- execute (cf. page précédente) si la procédure décrite dans le message M doit être exécutée localement ou
- send (cf. de la présente page) si cette procédure ne doit pas être exécutée localement.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de execute (cf. page 128) et send (cf. page précédente).

relation (cf. pages 85–91) ceux de execute (cf. page 128) et send (cf. page précédente).

remarque Tous les appels à execute (cf. page 128) ou send (cf. page précédente) doivent transiter par submit pour fournir une interface unifiée d'exécution de tous les messages.

submit_with_result Cette action correspond à l'action submit (cf. page précédente) mais avec un résultat. Elle a pour signature object submit_with_result(M : Message) et fait un choix entre execute_with_result (cf. page précédente), get_attribute_value (cf. page précédente) et send_with_result (cf. page précédente) au lieu de execute et send.

check_assertion

définition check_assertion exécute une assertion pour vérifier qu'elle est respectée.

signature boolean check_assertion(A : Assertion)
check_assertion exécute l'assertion A et rend true si A est respectée et false si A est violée.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de submit_with_result (cf. de la présente page).

relation (cf. pages 85–91) ceux de submit_with_result (cf. de la présente page).

remarques
- check_assertion fait généralement appel à submit_with_result (cf. de la présente page) pour évaluer A. Ceci est possible si nous considérons les assertions comme de simples expressions booléennes. Dans le cas contraire (si, par exemple, nous admettons un langage d'assertion distinct du langage de programmation), un processus d'évaluation spécifique devra être utilisé.
- check_assertion peut être appelé à tout moment, par exemple pour vérifier un invariant de description. Le méta-programmeur doit choisir ces moments, cette capacité n'étant pas paramétrable. Mais elle pourrait le devenir si le besoin s'en fait sentir.
- Si une assertion non vérifiée est rencontrée, une exception pourrait être levée, qui devra alors être traitée par tout appelant de check_assertion.

6.6 Actions de contrôle

Les actions suivantes permettent de réaliser des contrôles sur les applications écrites dans un concept-langage lui-même décrit en *OFL*. Ces contrôles peuvent, selon leur nature, être par exemple réalisés une fois pour toute par un compilateur ou effectués à chaque fois que nécessaire par un interprète ou une machine virtuelle. Ces

actions ont vocation à être utilisées soit directement soit par l'intermédiaire d'autres actions.

Nous avons séparé les actions de contrôle en deux catégories :

1. Celles qui débutent par le mot verify offrent au méta-programmeur, concepteur du modèle *OFL*, la possibilité de réaliser un certain nombre de vérifications structurelles. Il s'agit toujours de contrôler l'adéquation de l'application par rapport aux paramètres de concept-description et de concept-relation. Ces contrôles sont à la charge du méta-programmeur (ou du compilateur/interprète), nous ne signalons donc pas toujours quelles autres actions peuvent y faire appel. Il est très important de comprendre que, sauf dans le cas d'un langage très dynamique (avec création, modification et suppression de descriptions pendant l'exécution), il est inutile de lancer ces contrôles à plusieurs reprises. Ils vérifient l'intégrité structurelle (statique) de l'application et donnent donc toujours un résultat identique au fil de l'exécution.
2. La seconde partie des actions, celles qui ne commencent pas par verify, est constituée de contrôles et vérifications non structurels (donc souvent dépendants du déroulement des instructions à l'exécution) et parfois plus complexes.

verify_genericity

définition verify_genericity contrôle le bon usage de la généricité (c'est-à-dire l'adéquation des paramètres effectifs par rapport aux paramètres génériques formels des descriptions). Il s'agit de vérifier qu'une description n'est générique que si le paramètre Genericity le permet.

signature boolean verify_genericity()
Le résultat est true si le test ne révèle aucun problème, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
– Genericity pour vérifier la généricité de la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

verify_generator

définition verify_generator teste si la création d'objet est valide, c'est-à-dire si la description est en droit de générer des instances propres.

signature boolean verify_generator()
Le résultat est true si le test ne révèle aucun problème, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
– Generator pour vérifier la création d'instance par la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

remarque Une description qui ne peut générer d'instances propres (dont le paramètre de concept-description Generator est égal à false) peut tout de même décrire des constructeurs. Ceux-ci, comme ceux des classes abstraites de *Java* (cf. section 3.3.3 page 66), peuvent être appelés par d'autres descriptions pour créer des instances *indirectes* de la description courante.

is_valid_generation (cf. page 137) s'occupe du traitement de ce cas particulier (présence de constructeurs que nous ne pouvons appeler que dans certaines conditions) car ce contrôle n'est plus structurel. Il nécessite en effet de connaître le message d'appel au constructeur pour identifier l'appelant.

verify_destructor Cette action vérifie la validité de la destruction d'objets selon les mêmes principes que verify_generator (cf. page précédente), dont elle est le pendant. Elle met en œuvre le paramètre de concept-description Destructor au lieu de Generator.

verify_encapsulation

définition verify_encapsulation permet de vérifier que le choix de l'encapsulation des méthodes (dans la description ou en dehors de celle-ci) a bien été respecté.

signature boolean verify_encapsulation()
Le résultat est true si le test ne révèle aucun problème, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
– Encapsulation pour vérifier la politique d'encapsulation de la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

verify_attribute

définition verify_attribute teste si la présence d'attribut ne contredit pas une éventuelle interdiction d'en avoir dans la description.

signature boolean verify_attribute()
Le résultat est true si le test ne révèle aucun problème, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
– Attribute pour vérifier la présence d'attributs dans la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

verify_method Cette action est le pendant de verify_attribute (cf. de la présente page) pour les méthodes. Le paramètre de concept-description Method est ici utilisé plutôt que Attribute.

verify_overloading

définition verify_overloading contrôle que la politique de surcharge est correctement respectée, c'est-à-dire que la surcharge n'est présente que lorsqu'elle est permise.

signature boolean verify_attribute()
Le résultat est true si aucun problème n'est détecté, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
- Overloading pour vérifier la politique de surcharge de la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

verify_cardinality

définition verify_cardinality vérifie que la cardinalité de la relation concernée est conforme à celle qui est autorisée.

signature boolean verify_cardinality()
Le résultat est true si aucun problème n'est détecté, false sinon.

paramètres pris en compte dans une

description Cette action s'applique à une relation et n'est pas applicable pour une description.

relation (cf. pages 85–91)
- Cardinality pour contrôler la validité de la cardinalité de la relation.

verify_circularity

définition verify_circularity contrôle que la relation n'est directement ou indirectement circulaire que si cela est autorisé.

signature boolean verify_circularity()
Le résultat est true si la valeur du paramètre Circularity est bien respectée, false sinon.

paramètres pris en compte dans une

description Cette action s'applique à une relation et n'est pas applicable pour une description.

relation (cf. pages 85–91)
- Circularity pour contrôler la circularité de la relation.

verify_repetition

définition verify_repetition contrôle que la répétition dans les descriptions-sources ou dans les descriptions-cibles s'applique comme cela est prévu par le paramètre de concept-relation correspondant.

signature boolean verify_repetition()
Le résultat est true si aucun problème n'est détecté, false sinon.

paramètres pris en compte dans une

description Cette action s'applique à une relation et n'est pas applicable pour une description.

relation (cf. pages 85–91)
- Repetition pour contrôler la répétition des descriptions-sources ou descriptions-cibles pour cette la relation.
- Circularity pour tenir compte de la circularité de la relation et éviter ainsi d'entrer inutilement dans des cycles.

verify_variance

définition verify_variance contrôle que les déclarations de primitives et d'assertions dans une description sont compatibles avec la politique de variance définie par les paramètres correspondants du concept-relation.

signature boolean verify_variance()
Le résultat est true si aucun problème n'est détecté, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91)
- Feature_variance pour contrôler la variance des primitives et
- Assertion_variance pour vérifier celle des assertions.

verify_adaptation

définition verify_adaptation vérifie que l'adaptation des primitives (renommage, redéfinition, ...) se fait, au travers de la relation, conformément à ce qu'indique ses paramètres.

signature boolean verify_adaptation()
Le résultat est true si aucun problème n'est détecté, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91)
- Adding, Removing, Renaming, Redefining, Masking, Showing, Abstracting et Effecting pour contrôler les adaptations de primitives au travers de la relations.

remarques – Il est ici question de contrôler qu'aucune adaptation marquée forbidden n'est réalisée, mais aussi de vérifier que toutes les adaptations notées mandatory sont bien effectuées.
- Il est parfois nécessaire de vérifier la présence ou l'absence d'une primitive. Par exemple, si le paramètre Effecting indique mandatory, toutes les primitives abstraites des descriptions-cibles doivent être présentes (et concrètes) dans les descriptions-sources. La situation inverse est possible si Removing a pour valeur mandatory.

are_valid_parameters

définition are_valid_parameters teste la compatibilité entre les paramètres effectifs et formels (nombre et conformité de type) lors d'un appel de primitive.

signature boolean are_valid_parameters(M : Message, F : Feature)
M est le message représentant la requête et F la primitive candidate pour y répondre. Le résultat est true si les paramètres sont compatibles, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de are_valid_generic_parameters (cf. de la présente page) et is_conform_type (cf. de la présente page).

relation (cf. pages 85–91) ceux de are_valid_generic_parameters (cf. de la présente page) et is_conform_type (cf. de la présente page).

remarques

- are_valid_parameters fait généralement appel à are_valid_generic_-parameters (cf. de la présente page) pour tester la validité d'éventuels paramètres génériques et à is_conform_type (cf. de la présente page) pour contrôler la conformité de type de chacun des paramètres effectifs dans M en fonction du paramètre formel correspondant dans F.
- are_valid_parameters est un contrôle *a priori* des paramètres, donc sans les exécuter. Il s'agit de vérifier leur nombre et leur type. Cela limite grandement les risques d'erreur à l'évaluation (exécution) de ces paramètres mais ne garantit cependant pas un résultat sûr.

are_valid_generic_parameters

définition are_valid_generic_parameters teste la conformité des paramètres génériques[4] effectifs avec les paramètres génériques formels.

signature boolean are_valid_generic_parameters(M : Message, F : Feature)
Le résultat est true si le message M décrit un appel valide à une primitive F en ce qui concerne ses paramètres de généricité.

paramètres pris en compte dans une

description (cf. pages 79–81) en plus de ceux de is_conform_type (cf. de la présente page)

- Genericity pour vérifier que la description de F est effectivement générique.

relation (cf. pages 85–91) ceux de is_conform_type (cf. de la présente page).

remarques

- are_valid_generic_parameters fait généralement appel à is_conform_-type (cf. page 6.6) pour vérifier si le type de chaque paramètre générique effectif dans M est conforme au paramètre formel correspondant dans F.
- are_valid_generic_parameters peut éventuellement faire appel à verify_genericity (cf. page 6.6 page 131) pour vérifier que la description de F est bien générique. Dans ce cas, l'usage du paramètre de concept-description Genericity est superflu.

is_conform_type

définition is_conform_type teste la conformité entre deux types.

signature boolean is_conform_type(T1 : Type, T2 : Type)
T1 et T2 sont les deux types à comparer (pour vérification du type d'un paramètre effectif avec celui d'un paramètre formel correspondant, par exemple). Le résultat est true si T1 est conforme à T2, false sinon.

[4] Un paramètre générique est forcément un paramètre de description générique, jamais un paramètre de primitive.

paramètres pris en compte dans une

description (cf. pages 79–81)

– Visibility pour vérifier que la description de T2 est visible par T1.

relation (cf. pages 85–91)

– Kind pour suivre les relations, dans le graphe de types, à partir de T1 vers T2,
– Cardinality pour parcourir plusieurs branches du graphe d'importation en cas de relation multiple,
– Circularity pour éviter d'entrer dans des boucles infinies dans les relations circulaires,
– Opposite pour suivre les relations opposées de la cible vers la source et
– Polymorphism_implication et Polymorphism_policy pour calculer un graphe de types en fonction des relations polymorphiques.

remarques

– Par défaut, nous considérons que T1 est conforme à T2 si et seulement si T1 est égal à T2 ou est un sous-type de T2. De plus, nous disons que T1 est un sous-type de T2 dans le cas suivant : T1 (en fait la description dont T1 est un représentant) est la source et (la description de) T2 la cible directe ou indirecte d'une relation dont Polymorphism_implication est à up ou both.
– De plus, en ce qui concerne la généricité, T1 est conforme à T2 si et seulement si, T1 étant source et T2 cible, T1 a plus de paramètres génériques que T2 et ceux qui sont communs sont conformes.
– is_conform_type(T1, T2) est toujours équivalent à is_conform_type(T2, T1) si et seulement si
  – aucun concept-relation mis en œuvre n'admet de polymorphisme ou
  – tous les concept-relation mis en œuvre sont polymorphiques ascendants *et* descendants (correspond à la valeur both pour le paramètre de concept-relation Polymorphism_implication).
– is_conform_type peut éventuellement faire appel à is_same_type (cf. de la présente page) pour contrôler si T1 et T2 représentent exactement le même type.
– Le calcul du graphe de types peut éventuellement être partagé avec celui de local_lookup (cf. page 123) et lookup (cf. page 124).

is_same_type

définition is_same_type teste l'égalité entre deux types.

signature boolean is_same_type(T1 : Type, T2 : Type)

T1 et T2 sont les deux types à comparer. Le résultat est true si T1 et T2 sont identiques, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91) aucun.

remarques

– En l'état, is_same_type est une annexe pour is_conform_type (cf. page précédente) : elle permet, dans ce contexte, de tester l'égalité exacte

entre deux types avant de se lancer dans un parcours du graphe de types. Il peut être utile de redéfinir ces deux actions, et notamment is_same_type, dans le cas d'un langage où la notion de *même type* est plus complexe. Par exemple, nous pourrions dire que deux types représentés par les descriptions PERSONNE_V1 et PERSONNE_V2 unies par une relation *est-une-version-de* sont identiques au sens d'un langage incluant cette relation de version.

– Si le méta-programmeur admet une telle extension de la notion de *même type* dans un concept-langage, il lui sera alors utile, dans is_same_type, de mettre en œuvre les mêmes paramètres de concept-description et concept-relation que dans is_conform_type (cf. page 135).

is_valid_generation

définition is_valid_generation teste si un message appelant un constructeur est en droit de le faire.

signature boolean is_valid_generation(M : Message)
Le résultat est true si le message M décrit un appel valide à un constructeur, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81)
– Generator pour vérifier la validité de la création d'instance par la description.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

remarques
– is_valid_generation peut éventuellement faire appel à verify_generator (cf. page 6.6 page 131) sur la description du constructeur cité dans M (alors, l'usage du paramètre Generator devient ici inutile). Dans ce cas, si verify_generator rend true, is_valid_generation peut en faire autant. Si verify_generator rend false, is_valid_generation doit vérifier que nous ne nous trouvons pas dans un cas tout de même valide d'appel indirect (cf. la remarque dans verify_generator).
– is_valid_generation a donc pour seul but de vérifier, dans les cas où l'appel au constructeur cité dans M tente de créer une instance propre de la description de ce constructeur, que cela est permis.
– Le contrôle de la visibilité et des surcharges éventuelles du constructeur n'est pas dévolu à is_valid_generation mais à son appelant. Rappelons que lookup (cf. page 6.4 page 124), principale action utilisatrice (indirecte) de is_valid_generation prend en charge ces contrôles (toujours indirectement, par l'appel à match [cf. page 6.4 page 120]).

is_valid_destruction C'est le pendant de is_valid_generation pour les destructeurs. Il s'utilise de la même façon. is_valid_destruction met en œuvre le paramètre de concept-description Destructor au lieu de Generator.

is_valid_assertion

définition is_valid_assertion teste la validité d'une assertion.

signature boolean is_valid_assertion(A : Assertion)
Le résultat est true si l'assertion A est valide, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91)
- Assertion_Variance pour contrôler que la variance est respectée dans le cas d'une redéfinition.

remarques
- Si verify_variance (cf. page 6.6 page 134) a été activée sur la description de A alors il est inutile d'utiliser is_valid_assertion, le test de variance ayant déjà été fait.
- is_valid_assertion n'est pas indispensable dans l'état actuel des choses. Il nous a cependant semblé intéressant de le présenter en prévision de futurs paramètres. La recherche dans le domaine des assertions est active et les règles d'écriture de celles-ci ne le sont pas moins. Nous pourrions donc envisager de nouveaux paramètres pour gérer par exemple :
  - le droit de faire des effets de bord dans l'assertion,
  - le droit d'appeler des méthodes (procédures, fonctions, ...) dans l'assertion,
  - d'éventuelles restrictions d'usage des objets non instances de la description de l'assertion,
  - ...

6.7 Actions de gestion d'instance de description

La création et la destruction d'instances de description constituent des actions particulières qui sont généralement traitées de manière spécifique dans les langages à objets. Les actions de gestion d'instance de description permettent de mettre en œuvre ces opérations. L'existence des extensions de description est également fort utile pour les assertions.

before_create_instance

définition before_create_instance réalise toutes les tâches utiles avant la création d'une instance de description.

signature void before_create_instance(M : Message)
before_create_instance est, par exemple, chargé de réaliser, avant exécution du constructeur décrit dans M, le contrôle de la précondition du constructeur ainsi que l'évaluation et l'attachement des paramètres effectifs.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et aussi attach_parameters (cf. page 126).

relation (cf. pages 85–91) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et attach_parameters (cf. page 126).

remarques

- before_create_instance fait généralement appel à check_assertion (cf. page 130) pour évaluer chaque assertion de la précondition du constructeur. Il est également fait appel à evaluate_parameters (cf. page 125) puis à attach_parameters (cf. page 126) pour évaluer puis attacher les paramètres effectifs.
- La modification de l'objet courant n'est pas opportune ici car la construction d'une instance ne s'applique pas à un objet.
- Le contrôle de la validité de l'appel au constructeur est dévolu à match (cf. page 120) qui sera appelé *via* create_instance (cf. de la présente page).

after_create_instance

définition after_create_instance réalise les tâches utiles après la création d'une instance de description. C'est le pendant de before_create_instance (cf. page ci-contre).

signature void after_create_instance(M : Message)
after_create_instance est, par exemple, chargé de réaliser, après exécution du constructeur décrit dans M, le détachement des paramètres effectifs ainsi que le contrôle de la postcondition du constructeur et de l'invariant de description (si nécessaire). Enfin, il ne faut pas oublier d'ajouter l'instance dans l'extension de sa description.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130) et detach_parameters (cf. page 126).

relation (cf. pages 85–91) ceux de check_assertion (cf. page 130) et detach_parameters (cf. page 126).

remarque after_create_instance fait généralement appel à check_assertion (cf. page 130) pour évaluer chaque assertion atomique de la postcondition du constructeur et de l'invariant de description (s'il y a lieu). Il est également fait appel à detach_parameters (cf. page 126) pour détacher les paramètres effectifs.

create_instance

définition create_instance permet de créer (allouer et initialiser) une instance de description.

signature object create_instance(M : Message)
Le résultat est une nouvelle instance créée en fonction du message M qui doit faire appel à un constructeur.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de lookup (cf. page 124) before_create_instance (cf. page ci-contre) et after_create_instance (cf. de la présente page).

relation (cf. pages 85–91) ceux de lookup (cf. page 124), before_create_instance (cf. page précédente) et after_create_instance (cf. de la présente page).

remarques

- create_instance utilise généralement lookup (cf. page 124) pour rechercher le constructeur adapté. Il est fait appel à before_create_instance (cf. page 138). Puis l'allocation, l'initialisation et les autres tâches définies dans le constructeur appelé sont réalisées. Enfin, after_create_instance (cf. page précédente) est lancée.
- Par défaut, le polymorphisme ne s'applique pas aux constructeurs. Mais cela pourrait être adapté par la redéfinition de match (cf. page 120) voire par l'ajout d'un paramètre de concept-relation.
- create_instance peut être divisée en plusieurs sous-actions telles allocate_instance et initialize_instance si cela s'avère utile.
- Comme pour execute (cf. page 128), l'exécution de la construction de l'instance est soit gérée complètement par *OFL* soit déléguée au langage décrit par le concept-langage courant.

before_destroy_instance

définition before_destroy_instance réalise toutes les tâches utiles avant la destruction d'une instance de description.

signature void before_destroy_instance(M : Message)
before_destroy_instance est, par exemple, chargé de réaliser, avant exécution du destructeur décrit dans M, le contrôle de la précondition du destructeur et l'évaluation et attachement des paramètres effectifs. Il faut également retirer l'instance à détruire de l'extension de sa description propre.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et aussi attach_parameters (cf. page 126).

relation (cf. pages 85–91) ceux de check_assertion (cf. page 130), evaluate_parameters (cf. page 125) et attach_parameters (cf. page 126).

remarque before_destroy_instance fait généralement appel à check_assertion (cf. page 130) pour évaluer chaque assertion de la précondition du destructeur. Il est également fait appel à evaluate_parameters (cf. page 125) puis à attach_parameters (cf. page 126) pour évaluer puis attacher les paramètres effectifs.

after_destroy_instance

définition after_destroy_instance réalise toutes les tâches utiles après la destruction d'une instance de description. C'est donc le pendant de before_destroy_instance (cf. de la présente page).

signature void after_destroy_instance(M : Message)
after_destroy_instance est, par exemple, chargé de réaliser, après exécution du destructeur décrit dans M, le détachement des paramètres effectifs ainsi que le contrôle de la postcondition du destructeur et de l'invariant de description (si nécessaire). after_destroy_instance s'occupe également de détruire toutes les instances dépendantes de celle sur laquelle porte le destructeur.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de check_assertion (cf. page 130) et detach_parameters (cf. page 126).

relation (cf. pages 85–91) en plus de ceux utilisés par check_assertion (cf. page 130) et detach_parameters (cf. page 126)

– Dependence pour détruire toutes les instances dépendantes de celle sur laquelle porte le destructeur.

remarques

– after_destroy_instance fait généralement appel à check_assertion (cf. page 130) pour évaluer chaque assertion de la postcondition du destructeur et de l'invariant de description (s'il y a lieu). Il est également fait appel à detach_parameters (cf. page 126) pour détacher les paramètres effectifs.

– after_destroy_instance fait de plus appel à destroy_instance (cf. de la présente page) pour détruire toutes les instances dépendantes de celle qui vient d'être détruite (influence du paramètre de concept-relation Dependence), et ceci récursivement.

destroy_instance

définition destroy_instance permet de détruire (désallouer ou rendre au ramasse-miettes[5]) une instance de description.

signature object destroy_instance(M : Message)
Le résultat est la destruction de l'instance de description sur laquelle porte le destructeur cité dans M.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de lookup (cf. page 124), before_destroy_instance (cf. page ci-contre) et after_destroy_instance (cf. page précédente).

relation (cf. pages 85–91) ceux de lookup (cf. page 124), before_destroy_instance (cf. page précédente) et after_destroy_instance (cf. page ci-contre).

remarques

– destroy_instance fait généralement à l'action lookup (cf. page 124) pour rechercher le destructeur adapté. Il est ensuite fait appel à before_destroy_instance (cf. page ci-contre). Puis la désallocation ou la mise à la disposition du ramasse-miettes est réalisée. Enfin, after_destroy_instance (cf. page précédente) est lancée.

– Par défaut, nous n'appliquons pas de polymorphisme aux destructeurs.

6.8 Actions de gestion d'extension de description

Pour mieux définir la sémantique des relations et des descriptions, il est nécessaire de pouvoir paramétrer le fonctionnement de l'extension d'une description. Rappelons à ce propos que l'extension d'une description est l'ensemble des instances (directes et indirectes) de cette description.

[5] ramasse-miettes = *garbage collector*

create_extension

définition create_extension crée l'extension directe d'une description.

signature void create_extension(R : Primitive)
Le résultat est la création d'un ensemble d'objets affecté à l'extension directe d'une description. Cette extension peut être complétée par l'usage de R.

paramètres pris en compte dans une

description (cf. pages 79–81)
- Genericity pour tenir compte de l'éventuelle généricité de la description et
- Extension_creation pour choisir entre un méthode automatique (option par défaut) ou manuelle de la création de l'extension.

relation Cette action s'applique à une description et n'est pas applicable pour une relation.

remarques
- Plus précisément, create_extension crée une extension d'un type plutôt que celle d'une description. Pour une description non générique, c'est équivalent. Mais avec une description générique, plusieurs types peuvent être représentés et chacun possède sa propre extension : l'extension de la description elle-même est l'union des extensions des types effectifs décrits par cette description.
- L'extension gérée au niveau de chaque description est locale. Elle ne tient pas compte du polymorphisme qui permet de considérer un objet comme instance (indirecte) de plusieurs descriptions. Pour connaître l'extension complète d'une description, il faut utiliser calculate_extension (cf. de la présente page).
- Par défaut, nous considérons que chaque extension est créée dans la foulée de la création de la description. Elle est donc vide au départ.
- Si le paramètre de concept-description Extension_creation est à automatically, R est inutilisée et par défaut, l'extension est créée vide. S'il est à manually, create_extension crée une extension vide puis appelle R qui doit se charger de compléter l'extension. Le choix de passer ainsi une routine en paramètre de create_extension n'est donné qu'à titre indicatif et peut être remis en question selon l'implantation du modèle *OFL*, au profit d'une solution plus polymorphique par exemple.

calculate_extension

définition calculate_extension permet de calculer l'extension complète d'une description.

signature set of objects calculate_extension()
Le résultat est l'ensemble des instances, directes et indirectes, d'une description (en fait d'un type représenté par une description).

paramètres pris en compte dans une

description (cf. pages 79–81)
- Genericity pour tenir compte de l'éventuelle généricité de la description.

relation (cf. pages 85–91)
- Kind pour suivre les relations d'importation polymorphique,
- Cardinality pour parcourir plusieurs branches des graphes en cas de relation multiple,
- Circularity pour éviter d'entrer dans des boucles infinies dans les relations circulaires,
- Opposite pour éventuellement suivre les relations opposées de la cible vers la source et
- Polymorphism_implication pour déterminer les relations à suivre et dans quel sens.

remarques
- Soit une description D. Si D est la source d'une relation dont Polymorphism_implication est à up ou both, les instances de D sont aussi instance de la cible. Si D est la cible d'une relation dont Polymorphism_implication est à down ou both, les instances de D sont aussi instance de la source. En prenant garde aux éventuelles boucles, il est ainsi possible de calculer l'extension de toute description.

6.9 Opérations de base

Les dernières actions, dites de base, sont parmi les plus importantes. Il s'agit des différentes opérations qui peuvent s'appliquer sur toute instance de description, telles l'affectation, la comparaison, copie, ... Le méta-programmeur peut choisir de redéfinir ces actions essentielles et ainsi sélectionner une nouvelle sémantique pour ces opérations.

assign

définition assign permet de réaliser une affectation.

signature void assign(A : Attribute, O : object)
assign affecte l'objet O (source de l'affectation) dans l'attribut (ou la variable locale de méthode, assimilée à un attribut temporaire) A (cible de l'affectation).

paramètres pris en compte dans une

description (cf. pages 79–81) en plus de ceux de is_conform_type (cf. page 135)
- Sharing_control pour vérifier que O peut-être partagé.

relation (cf. pages 85–91) en plus de ceux de l'action is_conform_type (cf. page 135)
- Sharing_level pour vérifier que la relation d'utilisation décrivant l'attribut A admet le partage d'instance. Si ce n'est pas le cas, seul un objet O créé spécifiquement pour l'occasion peut être affecté à A.

remarques
- A et O doivent avoir préalablement été évalués. Pour cela, execute_with_result (cf. page 129) (ou get_attribute_value [cf. page 129]) est utile. Il est aussi utile de vérifier que A est bien un attribut (et non un simple résultat de fonction par exemple).
- assign fait généralement appel à is_conform_type (cf. page 135) pour contrôler que le type de O est conforme à celui de A.

– Si O ne peut être partagé ou si A n'admet pas le partage, une copie (par valeur) de O est créée et c'est elle qui est affectée à A.

reference_equality

définition reference_equality teste l'égalité de référence entre deux valeurs ou deux objets (c'est-à-dire que les deux objets doivent être les mêmes et non pas seulement posséder la même valeur).

signature boolean reference_equality(O1 : object, O2 : object)
reference_equality rend true si O1 et O2 sont le même objet, false sinon. Dans le cas où O1 et O2 sont en fait des valeurs de base et non des objets (au sens programmation par objets), c'est alors une égalité de valeurs qui est vérifiée.

paramètres pris en compte dans une

description (cf. pages 79–81) aucun.

relation (cf. pages 85–91) aucun.

contents_equality

définition contents_equality teste l'égalité de contenu entre deux objets (c'est-à-dire que les attributs de chacun des deux objets doivent être identiques deux à deux).

signature boolean contents_equality(O1 : object, O2 : object)
contents_equality rend true si O1 et O2 ont le même contenu, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de is_same_type (cf. page 136) et reference_equality (cf. de la présente page).

relation (cf. pages 85–91) ceux de is_same_type (cf. page 136) et reference_equality (cf. de la présente page).

remarques

– contents_equality fait généralement appel à l'action is_same_type (cf. page 136) pour contrôler, préalablement à la comparaison des attributs, que O1 et O2 sont de même type.
– Nous pourrions ajouter un paramètre de concept-description qui permettrait d'activer ou pas ce contrôle préalable de type. Il serait ainsi possible de modifier la sémantique opérationnelle de contents_equality sans la redéfinir, simplement par hyper-généricité. Nous avons fait le choix, arbitraire, de ne pas ajouter un tel paramètre car la redéfinition de contents_equality présente très peu de difficulté pour le méta-programmeur. Cette remarque est valable pour la plupart des comportements par défaut des opérations de base.
– contents_equality fait aussi appel à reference_equality (cf. de la présente page) pour vérifier l'égalité de chacun des attributs de O1 et O2.
– Si O1 et O2 sont des entités de base (les valeurs de type int de *Java* ou les objets expansés d'*Eiffel*, par exemple), le résultat de contents_equality doit être équivalent à celui de reference_equality (cf. de la présente page).

– Si contents_equality est à true alors reference_equality (cf. page précédente) aussi. Cela peut être contrôlé par une assertion structurelle.

contents_deep_equality

définition contents_deep_equality teste *en profondeur* l'égalité de contenu entre deux objets.

signature boolean contents_deep_equality(O1 : object, O2 : object)
contents_deep_equality rend true si O1 et O2 ont le même contenu en profondeur, false sinon.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de is_same_type (cf. page 136) et reference_equality (cf. page précédente).

relation (cf. pages 85–91) ceux de is_same_type (cf. page 136) et reference_equality (cf. page précédente).

remarques

– contents_deep_equality fait généralement appel à is_same_type (cf. page 136) pour contrôler, préalablement à la comparaison des attributs, que O1 et O2 sont de même type.
– contents_deep_equality s'appelle également récursivement pour vérifier l'égalité profonde de chacun des attributs de O1 et O2.
– contents_deep_equality fait enfin appel à reference_equality (cf. page précédente) pour vérifier l'égalité des attributs sur des entités de base (comme les ints de *Java*).
– Si O1 et O2 sont des entités de base (telles les valeurs de type int de *Java* ou les objets expansés d'*Eiffel*, par exemple), le résultat de contents_deep_equality doit être équivalent à celui de reference_equality (cf. page ci-contre) et à celui de contents_equality (cf. page précédente).
– Si contents_deep_equality est à true alors contents_equality (cf. page ci-contre) l'est également. Cela peut être contrôlé par une assertion structurelle.

copy

définition copy copie un objet.

signature void copy(O1 : object, O2 : object)
copy affecte chaque attribut de O2 dans celui correspondant de O1.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de is_conform_type (cf. page 135) et assign (cf. page 143).

relation (cf. pages 85–91) ceux de is_conform_type (cf. page 135) et assign (cf. page 143).

remarques

– copy fait généralement appel à is_conform_type (cf. page 135) pour contrôler que le type de O1 est conforme à celui de O2. Si ce n'est pas le cas, la copie est impossible.

– copy fait ensuite appel à assign (cf. page 143) pour affecter chaque attribut de O2 dans celui correspondant de O1.
– Il est indispensable que O1 soit déjà construit. Les valeurs des attributs de O1 avant la copie sont écrasées.
– Pour les entités de base et les objets non partageables copy réalise une simple copie de valeur conformément à ce que prévoit assign (cf. page 143).

deep_copy

définition deep_copy copie *en profondeur* un objet.

signature void deep_copy(O1 : object, O2 : object)
deep_copy copie chaque attribut de O2 dans celui correspondant de O1.

paramètres pris en compte dans une

description (cf. pages 79–81) en plus de ceux de is_conform_type (cf. page 135)
– Sharing_control pour vérifier si O2 et chacun de ses attributs sont partageables.

relation (cf. pages 85–91) en plus de ceux de l'action is_conform_type (cf. page 135)
– Sharing_level pour vérifier que les relations d'utilisation gérant les attributs de O1 admettent le partage.

remarques
– deep_copy fait généralement appel à is_conform_type (cf. page 135) pour vérifier la conformité de type entre O1 et O2. S'il n'y a pas conformité, la copie ne peut être réalisée.
– deep_copy s'appelle ensuite récursivement sur chaque attribut de O2. Cela permet de parcourir l'objet à copier. Lorsque une valeur de base ou un objet non partageable est rencontré, une simple copie de valeur est réalisée vers l'attribut correspondant de O1.
– Comme pour copy (cf. page précédente), il est indispensable que O1 soit déjà construit et cela de manière polymorphique à O2.
– deep_copy réalise une copie exacte à l'objet initial comme le fait copy (cf. page précédente). Cependant, contrairement à copy, la copie créée par deep_copy est totalement indépendante de l'original (pas seulement au premier niveau), c'est-à-dire qu'elle ne partage (quelque soit le niveau) aucun objet avec lui.

clone

définition clone clone un objet.

signature object clone(O : object)
clone crée un nouvel objet qui est une copie de O.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de create_instance (cf. page 139) et copy (cf. page précédente).

relation (cf. pages 85–91) ceux de create_instance (cf. page 139) et copy (cf. page précédente).

remarques
- clone fait généralement appel à create_instance (cf. page 139) pour créer l'objet résultat. Un message est pour cela construit reproduisant un appel au constructeur par défaut du type de O.
- clone fait ensuite appel à copy (cf. page 145) pour copier O dans l'objet résultat.
- clone est donc différent de copy (cf. page 145) dans le sens où il ne nécessite pas d'objet préexistant comme cible de la copie.
- Pour les entités de base et les objets non partageables, clone réalise, après création de la cible, une simple copie de valeur.

deep_clone

définition deep_clone clone *en profondeur* un objet.

signature object deep_clone(O : object)
deep_clone crée un nouvel objet qui est une copie *en profondeur* de O.

paramètres pris en compte dans une

description (cf. pages 79–81) ceux de create_instance (cf. page 139) et deep_copy (cf. page précédente).

relation (cf. pages 85–91) ceux de create_instance (cf. page 139) et deep_copy (cf. page précédente).

remarques
- deep_clone fait appel à create_instance (cf. page 139) pour créer l'objet résultat.
- deep_clone fait ensuite appel à deep_copy (cf. page précédente) pour copier O dans l'objet résultat.
- deep_clone est différent de l'action deep_copy (cf. page ci-contre) de la même manière que clone (cf. page précédente) l'est de copy (cf. page 145).
- Pour les entités de base et les objets non partageables, deep_clone réalise, après création de la cible, une simple copie de valeur.

6.10 Exemple de pseudo-code de l'action lookup

Nous souhaitons ajouter à la liste d'actions ci-dessus un exemple de pseudo-code pour l'une d'entre elles. Nous avons choisi l'action lookup qui tient compte, comme cela est décrit section 6.4 page 124, de la valeur du paramètre de concept-relation Polymorphism_implication.

```
// Action de composant-description : lookup
Feature lookup(M : MESSAGE)
  // Variables locales
  Feature F ;
  set of Feature SF ;
  // Corps
  F ← null ;
  // Création du graphe de type global
  // cf. un exemple de pseudo code de computeTypeGraph après celui de lookup
```

```
  computeTypeGraph() ;
  // Recherche locale, dans la description courante, de la primitive
  F ← local_lookup(M) ;
  // Une primitive locale convient-elle ?
  // Le présent algorithme privilégie une primitive locale et ne recherche hors de
  // la description courante que si aucune primitive locale ne correspond.
  // Ce choix non traditionnel montre une possibilité d'adaptation d'un langage.
  if F != null
  then
    // Une primitive locale convient : la recherche est terminée
    return F
  else
    // Aucune primitive locale ne convient
    // Recherche globale, en tenant compte des relations d'importation
    // cf. action all_features section 6.4 page 121
    SF ← all_features() ;
    if SF.isEmpty() ;
    then
      // Aucune primitive n'est candidate : la recherche est infructueuse
      return null
    else
      // Parcours de SF pour trouver une primitive qui convienne
      F ← SF.first() ;
      // cf. action match section 6.4 page 120
      while not (SF.isEmpty() or match(F, M))
        F ← SF.next() ;
      end while
      if match(F, M)
      then
        // Une primitive non locale convient : la recherche est terminée
        return F
      else
        // Aucune primitive ne convient
        return null
      end if
    end if
  end if
end function

// Procédure auxiliaire : computeTypeGraph
// Calcul du graphe de types GT de l'application courante
// Ce calcul n'est effectivement réalisé qu'au premier appel
// sauf si le graphe est dynamique
void computeTypeGraph()
  // Variables locales
  Type T, T2 ;
  Relationship R ;
  // Corps
  // Soit ST l'ensemble des types de l'application courante
  // Création des nœuds de GT
```

```
 for all T in ST
  GT.addNode(T) ;
 end for
 // Parcours de GT pour créer les arcs
 for all T in GT
  // cf. l'atome Type section 5.2.6 page 113
  for all R in T.description.sourceRelationships
   // cf. les paramètres de concept-relation section 4.4.2 page 85
   if R.Kind == import and
    (R.Polymorphism_implication == up or R.Polymorphism_implication == both)
   then
    // Création d'un arc de la source vers la cible de la relation
    // cf. l'atome Relationship section 5.2.9 page 115
    for all T2 in R.targetDescription
     GT.createArc(T, T2) ;
    end for
   end if
  end for
 end for
 // Ceci n'est qu'un exemple possible :
 // les polymorphismes down ne sont pas pris en compte.
end procedure
```

6.11 Bilan

Le présent chapitre vous a présenté le système d'actions qui constituent, avec les paramètres, l'épine dorsale du modèle *OFL*. Les paramètres indiquent le comportement souhaité au travers de valeurs simples et les actions sont les algorithmes (la sémantique opérationnelle) utiles pour que les indications des paramètres soient respectées. *A priori*, sauf modification ultérieure du modèle lui-même, la liste des actions est fixe et ne peut être modifiée. Par contre, des évolutions, dans le cadre des futurs travaux de recherche sur *OFL* permettront assurément de préciser et d'enrichir ce mécanisme de manière significative.

Nous avons vu tout au long de ce chapitre que des pistes font justement apparaître d'éventuels nouveaux paramètres hyper-génériques. En voici une compilation :

- Nous pourrions ajouter un paramètre de concept-description qui indiquerait si le passage des paramètres de méthode doit se faire par valeur ou par référence. Cela entraînerait notamment une modification du code de l'action attach_parameters (cf. section 6.5 page 126). La politique par défaut déterminée grâce à ce paramètre hyper-générique pourrait être complétée par des qualifieurs associés aux paramètres de méthode.
- L'ordre, voire la politique d'évaluation de ces mêmes paramètres de méthode pourrait également faire l'objet d'un paramétrage hyper-générique. Cela influencerait le code de l'action evaluate_parameters (cf. section 6.5 page 125).
- La stratégie d'activation des assertions pourrait elle-aussi être précisée à l'aide d'un paramètre hyper-générique. Nous pourrions même imaginer trois paramètres définissant séparément cette stratégie pour les préconditions, postconditions et invariants de description. L'ajout de ce paramètre aurait pour conséquence de revoir le code de l'action check_assertion (cf. section 6.5 page 130).

De plus, comme la stratégie d'activation, le comportement à appliquer en cas de violation d'une assertion peut être paramétré.
- D'autres caractéristiques concernant les assertions pourraient faire l'objet de paramètres. Citons par exemple la possibilité ou non de réaliser des effets de bord dans les assertions ou encore le contrôle voire le filtrage de l'appel aux primitives, de l'accès aux attributs dans les assertions. Tout cela pourrait être pris en compte lors de l'appel à check_assertion (cf. section 6.5 page 130).
- L'application du polymorphisme aux constructeurs, initialiseurs, destructeurs et, plus généralement, à toute primitive particulière, ou encore leur politique de chaînage, pourrait également être à l'origine de plusieurs paramètres hyper-génériques. Cela pourrait entraîner des modifications dans les actions create_instance (cf. section 6.7 page 139), destroy_instance (cf. section 6.7 page 141), execute (cf. section 6.5 page 128) et execute_with_result (cf. section 6.5 page 129).
- Un paramètre hyper-générique peut aussi être utile à l'action contents_equality (cf section 6.9 page 144) pour indiquer si un contrôle de type doit précéder le contrôle de valeur lors d'une comparaison sur cette dernière.
- ...

Chacune des idées, chacun des comportements ci-dessus pourrait donc donner lieu à un ou plusieurs paramètres hyper-génériques. Les listes de paramètres que nous avons données dans les sections 4.3.2 page 79 et 4.4.2 page 85 sont donc amenées à évoluer, à être complétées.

Il faut cependant définir un compromis entre le nombre de comportements paramétrés et la complexité du code des actions. En effet, plus nous ajoutons de paramètres, plus nous permettons aux méta-programmeurs d'adapter la sémantique opérationnelle de ses composants de langage. Mais, dans le même temps, nous rendons les actions de plus en plus complexes puisqu'elles tiennent compte de ces paramètres. La différence entre les paramètres que nous avons intégrés et ceux que nous citons ci-dessus comme possibles — il y en a bien d'autres — relève donc de ce choix arbitraire et pragmatique, choix que l'expérience d'utilisation d*OFL* nous fera probablement adapter.

Chapitre 7

OFL : exemples, apports et outils

7.1 Exemples de relation inter-description définie en *OFL*

Cette section présente quelques relations inter-descriptions définies en *OFL*. Ainsi que nous l'avons indiqué, notre travail s'est relativement focalisé sur les relations d'importation qui furent à l'origine du sujet étudié. Nous proposerons tout d'abord des relations traditionnelles, telles qu'il est possible de les trouver dans les langages à objets courants. Nous suggérons ensuite d'autres relations, plus originales, de celles que nous aimerions proposer aux programmeurs.

Pour chacune des relations utilisées en exemple, nous fournissons une valeur pour chaque paramètre de concept-relation. Nous commentons aussi ces choix en indiquant leurs incidences à chaque fois que cela nous semble intéressant et envisageons, éventuellement, des alternatives à ce choix.

7.1.1 Relations traditionnelles

Nous présentons ici deux relations classiques : la spécialisation et l'implémentation.

Spécialisation

La Spécialisation décrit la relation entre une description-source et une description-cible, la première spécialisant la seconde. Elle est souvent mise en œuvre, dans les langages à objets, par le mécanisme d'héritage, ce dernier se révélant être plus permissif que ne le veut la spécialisation. Nous disons par exemple que la description Carré spécialise la description Rectangle ou que Employé spécialise Personne.

Mettons nous donc à la place du méta-programmeur qui souhaite créer une telle relation pour la fournir au sein d'une bibliothèque d'*OFL*-composants et voyons quelle valeur nous allons choisir pour chaque paramètre de concept-relation (revenez à la section 4.4.2 page 85 pour la liste commentée de ces paramètres) :

Name Ici, c'est facile, nous prenons "Specialisation".

Kind La spécialisation est une relation inter-description d'importation. Nous prenons donc la valeur import.

Context Nous souhaitons réaliser un composant de bibliothèque : la valeur de Context sera donc library. Nous ne pouvons donc complètement renseigner les caractéristiques validSourceDescriptionConcepts et validTargetDescriptionConcepts du composant-relation Specialisation puisque nous ne pouvons connaître ici l'ensemble des composants-descriptions disponibles.

Services Ce paramètre (en fait un ensemble de paramètres) est valué à l'ensemble vide, notre problème n'étant pas ici d'intégrer une nouvelle classe de service (cf. section 8.1 page 175).

Cardinality Nous choisissons de créer une relation simple et non multiple. La valeur de Cardinality sera donc $1-1$. La définition d'une relation multiple aurait eu une incidence sur le paramètre suivant (Repetition).

Repetition Nous avons affaire à une relation simple, la valeur de Repetition sera donc de toutes façons ignorée.

Circularity La circularité doit être interdite pour la relation de spécialisation. En effet, il est évident que si une description A spécialise une description B, cette dernière ne peut, ni directement ni indirectement, spécialiser A. Nous pourrions tolérer que A spécialise directement A (sans effectuer aucune modification lors de cette spécialisation) mais cela n'apporterait rien. La valeur de Circularity sera donc forbidden.

Symmetry La spécialisation n'a rien de symétrique, nous sélectionnons donc ici la valeur false.

Opposite De manière générale dans les langages à objets courants, la spécialisation (usage le plus fréquent de l'héritage) ne connaît pas de relation opposée. Ici, nous en décrivons pourtant une, nommée évidemment *généralisation* : vous pouvez en trouver une définition dans la section 7.1.2 page 156. Cependant, nous indiquons ici none pour la valeur de Opposite. En effet, il ne faut pas oublier que nous nous trouvons dans un contexte de bibliothèques. Nous ne savons donc pas quels sont les autres composants-relations qui seront finalement présents dans le langage utilisé : la généralisation de la section 7.1.2 page 156 pourrait être absente voire remplacée par une autre.

Direct_access Une grande partie de l'intérêt d'une relation de spécialisation repose sur le fait que les instances de la description spécialisante disposent automatiquement et de manière transparente des primitives de la description spécialisée. Direct_access prend donc ici la valeur allowed.

Indirect_access Nous pensons que l'accès indirect dans la description spécialisante aux primitives de la description spécialisée (par exemple, par nommage explicite de la description spécialisée ou par l'usage d'un mot-clé super) n'a aucune raison d'être limité. Cette capacité ne réduit en rien les avantages de l'accès direct et est bien pratique dans des cas fréquents. Indirect_access comme Direct_access est donc valué à allowed ; les deux sortes d'accès sont donc autorisées.

Polymorphism_implication Le sens du polymorphisme de la spécialisation va de la description spécialisante vers la description spécialisée. C'est-à-dire que toutes les instances de la description spécialisante sont aussi instances (indirectes) de la description spécialisée. La valeur de Polymorphism_implication est donc up.

Polymorphism_policy Polymorphism_implication étant différent de none, la valeur de Polymorphism_policy sera prise en compte. Pour notre spécialisation, nous

choisissons d'appliquer une politique de redéfinition aussi bien aux méthodes qu'aux attributs. Nous rejoignons ainsi le choix de l'héritage d'*Eiffel* plutôt que de celui de *Java* où les attributs sont remplacés au lieu d'être redéfinis. La valeur choisie est donc <overriding, overriding>.

Feature_variance La variance est un autre point fondamental de l'héritage et donc de la spécialisation. La valeur de Feature_variance est un triplet indiquant la variance pour les paramètres de méthode, les résultats de fonction et les attributs. Notre choix est conforme à une spécialisation des plus classiques : <covariant, nonvariant, covariant>. Les paramètres de méthode et les attributs devront vérifier la covariance, ce qui est particulièrement pratique. Quant aux résultats de fonction, comme c'est souvent le cas, nous choisissons de les conserver invariants. La covariance serait ici également envisageable : nous assimilerions alors, par exemple, le résultat de fonction à un paramètre résultat de procédure.

Assertion_variance Toujours à propos de variance, il nous faut maintenant déterminer comment réagissent les assertions. Ici également, nous allons choisir le comportement traditionnel de celles-ci qui se retrouve par exemple dans *Eiffel*. Nous avons donc des préconditions allégées et des postconditions et invariants renforcés dans la description spécialisante par rapport à ceux de la description spécialisée. Ce qui nous donne la valeur <strengthened, weakened, strengthened> (rappelons que le triplet s'applique dans l'ordre à invariant puis précondition et enfin postcondition) pour le paramètre Assertion_variance.

Adding La spécialisation permet l'ajout de primitives, nous choisissons donc ici la valeur allowed.

Removing Nous signalons l'interdiction de supprimer des primitives lors de l'importation par la valeur forbidden pour ce paramètre. Le comportement standard de la spécialisation indique en effet que les objets plus spécialisés doivent savoir faire au moins tout ce que font les objets moins spécialisés. En adéquation avec les valeurs de Polymorphism_implication, Polymorphism_policy et Feature_variance, cette valeur de Removing impose de conserver toutes les primitives. Celles-ci peuvent cependant être adaptées (cf. les paramètres ci-dessous).

Renaming Nous souhaitons permettre le renommage et choisissons donc la valeur allowed pour ce paramètre. Il est en effet pratique de pouvoir donner, au sein de la description spécialisante, un nom plus spécifique à une primitive qui, issue de la description spécialisée, définit un comportement plus précis. Il peut s'agir d'un simple renommage pour s'adapter au contexte (par exemple, la description Pilote spécialise la description Conducteur et renomme la primitive conduire en piloter) ou d'un renommage accompagné d'une redéfinition ou concrétisation (cf. ci-dessous pour ces dernières ; par exemple, la description Bateau spécialise la description abstraite Véhicule et renomme et concrétise la primitive déplacer en naviguer.)

Redefining La redéfinition est une des caractéristiques les plus intéressantes de la spécialisation. Elle permet en effet de modifier la définition d'une primitive spécialisée et laisse ainsi le champ ouvert à la liaison dynamique. Le paramètre Redefining constitue un quadruplet de valeurs décrivant les droits de redéfinition pour les assertions, la signature, le corps et enfin les qualifieurs de primitive. Pour la spécialisation, nous autorisons (valeur allowed) la redéfinition des assertions (en adéquation avec la politique définie par Assertion_variance),

la signature (en relation avec Feature_variance) et, enfin, le corps des primitives. Nous permettons également la redéfinition des qualifieurs mais laissons ici la responsabilité au méta-programmeur d'écrire des assertions *OFL* pour la contrôler.

Masking Pour rester dans l'idée que la spécialisation implique de conserver au moins les comportements de la description spécialisée, nous décidons d'interdire le masquage et choisissons donc la valeur forbidden pour Masking. Nous faisons ici un choix arbitraire en sélectionnant la valeur qui nous semble la plus adaptée. Le méta-programmeur est souvent amené à faire un tel choix qui assure la cohérence de ces créations.

Showing Si le masquage est interdit, nous ne voyons par contre aucun inconvénient à autoriser le démasquage ; cela ne remet en effet nullement en cause l'idée de spécialisation. La valeur de Showing est donc allowed pour ce composant-relation. Les démasquages pourraient cependant être rares puisque les primitives ne pourront avoir été masquées que par une autre relation que la spécialisation.

Abstracting Par spécialisation, une primitive concrète ne peut être rendue abstraite d'où la valeur forbidden pour ce paramètre.

Effecting Par contre, il est tout à fait légitime de concrétiser dans la description spécialisante une primitive qui est abstraite dans la description spécialisée. C'est même un cas des plus fréquents, les descriptions abstraites étant souvent, par essence, plus générales que les descriptions concrètes. Il serait cependant exagéré d'obliger à concrétiser les primitives puisque la spécialisation se définit aussi bien entre descriptions abstraites. La valeur choisie est donc allowed.

Voilà qui décrit, sous forme de paramètres, les principaux comportements (et donc la sémantique opérationnelle) d'une très traditionnelle relation de spécialisation. Nous n'avons pas étudié pour cet exemple, comme pour les suivants, les paramètres spécifiques aux relations d'utilisation. Ceux-ci n'ont aucune incidence sur les relations que nous définissons.

Implémentation

L'Implémentation est aussi une relation couramment utilisée. Elle sert notamment dans *Java* à concrétiser dans une classe les définitions données dans une interface. Toujours dans notre rôle de méta-programmeur, enrichissons donc notre bibliothèque de relations inter-descriptions de l'implémentation. Voici nos choix pour les paramètres. Nous commentons les décisions les plus intéressantes.

Name "Implementation".

Kind import.

Context library.

Services <>.

Cardinality Sur le modèle du implements de *Java*, nous souhaitons offrir une relation multiple. Il n'y a *a priori* aucune raison de limiter le nombre de descriptions-cibles, la valeur de Cardinality est donc ici $1 - \infty$.

Repetition Implémenter deux fois une description abstraite n'offre aucun intérêt pratique. D'un autre côté, aucun inconvénient ne l'empêche puisqu'une description

abstraite est forcément compatible avec elle-même. Cela n'amenant rien, nous choisissons d'interdire la répétition dans les descriptions-cibles. La valeur de Repetition est donc <*ignoré*, forbidden> (pour la première valeur concernant la source, qui doit être unique [cf. Cardinality ci-dessus], la valeur est ignorée).

Circularity L'implémentation peut consister à rendre concrètes toutes les primitives abstraites importées ou seulement une partie. Notre choix se porte ici sur la première solution (mais rien n'empêchera le méta-programmeur que nous sommes de fournir aussi une relation Implémentation_Partielle). Dans cette idée d'implémentation complète, la relation Implémentation ne peut être circulaire. La valeur de Circularity sera donc forbidden.

Symmetry false.

Opposite none.

Direct_access Dans la logique de notre relation, toutes les descriptions-cibles sont intégralement abstraites (comme des interfaces en *Java*). La description-source se voit donc contrainte (nous le verrons encore mieux plus bas dans le paramètre Effecting) de donner un corps à chaque primitive importées. Ainsi, fait relativement rare pour une relation d'importation, l'accès direct aux primitives importées perd grandement de son intérêt. Nous choisissons donc la valeur forbidden pour ce paramètre.

Indirect_access Pas plus l'accès indirect que l'accès direct ne nous semble présenter une quelconque utilité au travers d'une relation d'implémentation. La valeur sélectionnée est donc aussi ici forbidden.

Polymorphism_implication Le sens du polymorphisme d'une relation d'implémentation est identique à celui d'une spécialisation. Intuitivement, nous disons en effet que toutes les instances de la description concrète sont aussi instances de la description abstraite. La valeur de Polymorphism_implication est donc up.

Polymorphism_policy Là encore, nous n'avons pas de raison de nous éloigner de la sémantique de la spécialisation. Et cela paraît cohérent : l'implémentation telle que nous la définissons n'est, après tout, qu'une forme particulière de spécialisation accompagnée d'une concrétisation obligatoire. La valeur choisie est donc <overriding, overriding>.

Feature_variance Toujours dans le prolongement de la spécialisation : <covariant, nonvariant, covariant>.

Assertion_variance <strengthened, weakened, strengthened>.

Adding allowed.

Removing Le but est de concrétiser un comportement pas de le réduire, nous sélectionnons donc forbidden pour ce paramètre.

Renaming La capacité de renommage est bien pratique et ne gêne en rien. allowed est donc la valeur de Renaming. Rappelons que le renommage est une caractéristique intéressante mais que nous pouvons qualifier de purement syntaxique. Un nom plus adapté pour une primitive est un atout évident mais cela ne remet pas en cause la liaison que la primitive avec le nouveau nom possède avec la primitive importée ou utilisée. La liaison dynamique notamment y voit bien la même entité, comme cela est réalisé en *Eiffel*.

Redefining La redéfinition peut être autorisée pour les assertions (cf. Assertion_variance), la signature (cf. Feature_variance) et les qualifieurs des primitives.

Par contre, elle est interdite pour les corps car les primitives importées sont toutes abstraites et ne possèdent donc pas de corps. La valeur de Redefining est donc <allowed, allowed, forbidden, allowed>. Précisons que toute redéfinition doit obligatoirement s'accompagner d'une concrétisation (cf. Effecting ci-après).

Masking forbidden.

Showing Ici, notre choix est beaucoup plus arbitraire. La description abstraite a toutes les chances de ne contenir aucune primitive masquée puisqu'elle représente une interface (au sens *liste de caractéristiques*). Cependant, aucun argument ne nous semble requérir l'interdiction de la capacité de démasquage au travers d'une implémentation[1]. La valeur que nous sélectionnons pour Showing est donc allowed.

Abstracting Abstracting a ici la valeur forbidden : toutes les primitives importées sont abstraites, tenter de les abstraire n'a donc aucun sens.

Effecting Voilà maintenant un des paramètres les plus importants pour la relation d'implémentation telle que nous la définissons. Rappelons le contexte : une description concrète (sans partie abstraite donc) donne une implémentation à *toutes* les primitives d'une description abstraite. La première est une sorte de spécialisation concrète de l'abstraction que figure la seconde. Pour que tout cela fonctionne selon ce principe, il nous faut choisir la valeur mandatory pour le paramètre Effecting. Ainsi, le programmeur qui utilise une relation instance du composant-relation Implémentation se verra contraint de donner un corps à chaque méthode. De plus, Il pourra, comme cela est spécifié par les paramètres précédents, renommer et/ou redéfinir et/ou démasquer les primitives qu'il concrétise.

7.1.2 Relations moins courantes

Après avoir défini les relations couramment utilisées dans les langages à objets à classes, dont celles ci-dessus, le méta-programmeur souhaite désormais décrire des relations moins traditionnelles. Ainsi, *OFL* lui permet de mettre à la disposition du programmeur des composants d'un langage adaptés à ses besoins.

Nous présentons donc maintenant quelques relations qui sont actuellement mises en œuvre par héritage mais qui gagnent à être spécifiées plus précisément.

Généralisation

Dans la section 7.1.1 page 151, nous avons défini une relation de spécialisation, la plus couramment recherchée lorsqu'un héritage est réalisé. La relation inverse de la spécialisation est la généralisation. Son usage peut toujours être évité et remplacé par une spécialisation dans le sens inverse. Cependant, il existe de nombreux cas où l'existence de la généralisation, en plus de son inverse, serait un atout. Prenons un exemple : une application est réalisée. Son schéma de descriptions (classes) est stable et bien conçu, il répond à toutes les attentes. Plus tard, cette application doit évoluer. Pour une des raisons suivantes par exemple, modifier les descriptions initiales n'est pas envisageable :

[1] À propos du démasquage, nous rappelons qu'il s'agit de rendre visible (et donc utilisable) dans la description-source, une primitive masquée (et donc présente mais non utilisable) dans la description-cible.

– Nous ne possédons pas les codes sources des descriptions initiales. C'est en fait souvent le cas lorsque nous utilisons les bibliothèques standards d'un langage de programmation.
– Nous possédons les codes sources mais ils sont sous licence et donc non modifiables.
– Nous possédons les codes sources mais ils sont partagés par plusieurs applications et nous ne voulons pas prendre le risque de briser la fiabilité et/ou la robustesse de toutes ces dernières par un processus d'évolution qui ne concerne que l'une d'entre elles. C'est un cas fréquent avec les bibliothèques.

Illustrons cela par une hiérarchie volontairement simpliste : une bibliothèque de descriptions en contient deux : Container et ListeDoublementChaînée qui spécialise Container.

Container est abstraite et définit trois méthodes :

1. void clear() vide le container,
2. boolean isEmpty() indique si le container est vide ou non et
3. integer size() donne le nombre d'objets contenus par le container.

ListeDoublementChaînée est concrète et ajoute aux trois primitives précédentes :

1. void first() pour se placer sur le premier élément de la liste,
2. void last() pour se placer sur le dernier élément de la liste,
3. void next() pour avancer d'un élément,
4. void previous() pour reculer d'un élément et
5. Object getElement() pour consulter l'élément courant.

Vous l'aurez sans doute compris, nous aimerons ajouter une description ListeSimplementChaînée à cette hiérarchie *sans remettre en cause et donc modifier* les descriptions initiales.

ListeSimplementChaînée spécialiserait Container. Elle décrit les primitives suivantes.

1. void first() pour se placer sur le premier élément de la liste,
2. void next() pour avancer d'un élément et
3. Object getElement() pour consulter l'élément courant.

La spécialisation de Container ne pose aucun problème car elle est déclarée par ListeSimplementChaînée. La généralisation de ListeDoublementChaînée n'en poserait pas plus si elle était possible sans modifier cette dernière et donc sans prendre de risque avec les descriptions qui l'importent ou l'utilisent.

La figure 7.1 page suivante[2] présente l'intérêt de posséder une relation de généralisation pour enrichir un graphe de descriptions *sans modifier la hiérarchie initiale* mais *avec un graphe final fidèle à la modélisation*.

Étudions maintenant les valeurs à associer aux paramètres de concept-relation pour créer un composant-relation de Généralisation.

Name "Generalisation".

Kind import.

Context library.

[2]Rappelons que la légende des figures pour tout ce document est donnée dans la figure 3.1 page 48.

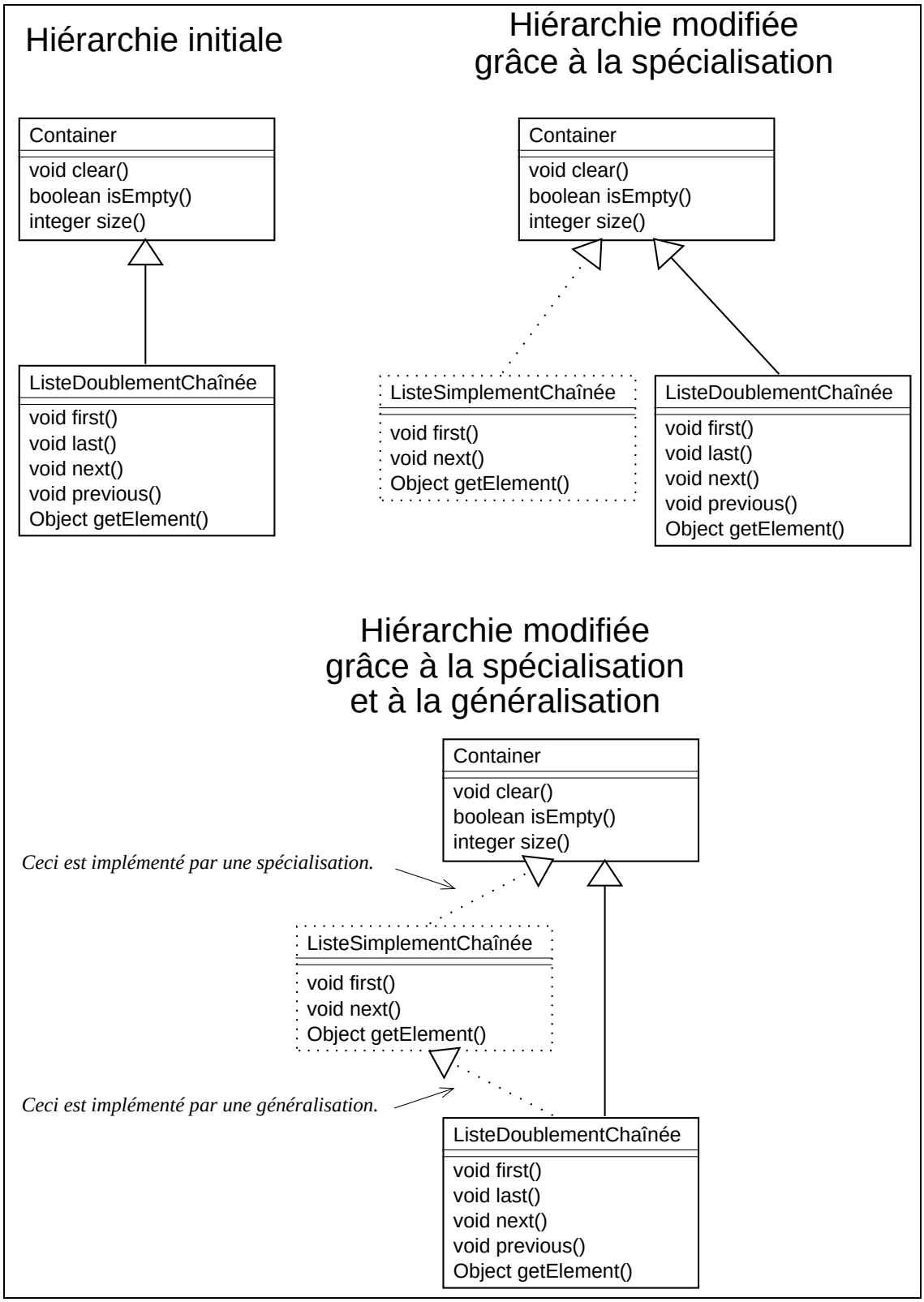

FIG. 7.1 – Exemple d'apport d'une relation de généralisation

Services <>.

Cardinality La spécialisation définie précédemment (cf. section 7.1.1 page 151) étant simple, nous définissons une généralisation sur le même choix : $1-1$.

Repetition Cette valeur est ignorée pour une relation de cardinalité $1-1$.

Circularity forbidden.

Symmetry false.

Opposite Ici, nous avons la possibilité d'indiquer le composant-relation Spécialisation comme inverse de de Généralisation. Si nous adoptons ce choix, cela améliorera sensiblement les fonctionnalités du graphe de descriptions (généralisations et spécialisations se mettront à décrire la même relation sémantique au sens près). Cependant, il nous faudra alors logiquement modifier le paramètre Opposite de Spécialisation pour y indiquer Généralisation.

Direct_access allowed (cohérent avec Spécialisation).

Indirect_access allowed (cohérent avec Spécialisation).

Polymorphism_implication Par essence, la polymorphisme de la généralisation est l'inverse de celui de la spécialisation, c'est donc down qu'il faut choisir ici.

Polymorphism_policy <overriding, overriding> (cohérent avec Spécialisation).

Feature_variance Ici aussi, pour bien spécifier une relation opposées à la spécialisation, nous n'avons qu'à inverser la valeur du paramètre Feature_variance et donc avoir <contravariant, nonvariant, contravariant>. Les résultats de fonction devront rester invariants alors que les paramètres de méthode et les attributs appliquerons une règle de contravariance.

Assertion_variance Logiquement, nous inversons les valeurs des paramètres de la spécialisation et trouvons <weakened, strengthened, weakened>, c'est-à-dire que les postconditions et les invariants doivent être identiques ou allégés et les préconditions être identiques ou renforcées.

Adding Au contraire de la spécialisation, et selon le principe qu'un objet spécialisé doit être en mesure d'effectuer toutes les opérations d'un objet plus général, il n'est pas envisageable de permettre l'ajout de primitive dans une généralisation. La valeur de ce paramètre est donc forbidden.

Removing Nous ne pouvons pas ajouter de primitive lors d'une généralisation mais nous allons évidemment permettre d'en supprimer. C'est donc allowed qui est sélectionné ici.

Renaming Les capacités de renommage sont équivalentes que nous soyons en train de spécialiser ou de généraliser une description. La valeur allowed est donc appropriée.

Redefining <allowed, allowed, allowed, allowed> est une valeur valide, la redéfinition des assertions, des signatures, des corps et des qualifieurs de primitives est donc permise. Il sera tenu compte lors de ces redéfinitions de la politique de variance des composants de primitives et des assertions.

Masking La spécialisation interdit le masquage et autorise le démasquage, la généralisation agit à l'inverse. Masking prend donc la valeur allowed.

Showing forbidden (cohérent avec Spécialisation).

Abstracting Généraliser c'est souvent abstraire. Abstracting prend donc la valeur allowed.

Effecting Si une primitive est abstraite dans la description à généraliser, elle ne peut pas être concrète dans la description généralisante. forbidden est donc adapté ici.

Nous disposons maintenant de la relation inverse à la spécialisation : la généralisation. D'un usage plus marginal, elle n'en est pas moins intéressante notamment pour faire évoluer une hiérarchie sans modifier le code existant.

Réutilisation de code

L'héritage est souvent utilisé pour mettre en œuvre une spécialisation. Pourtant, combien de fois n'en avons nous pas fait usage à d'autres fins, par exemple dans le but de réutiliser du code. C'est une amélioration particulièrement significative par rapport au copier/coller du code car elle ouvre la voie à une réutilisation accrue. Cependant, utiliser l'héritage dans ce but, c'est rencontrer des contraintes gênantes comme :

- laisser la place au polymorphisme là où il n'a pas de raison d'être,
- ne pas pouvoir choisir les méthodes à réutiliser mais devoir toutes les importer,
- être contraint par les règles d'adaptation des primitives (renommage, redéfinition, ...), règles qui sont souvent une gêne dans ce contexte,
- ...

Pour mieux comprendre ce que pourrait donner une capacité de réutilisation de code, imaginons les descriptions Personne, Automobile suivantes[3].

```
class Personne {
  integer annéeDeNaissance ;
  integer âge() {
    return annéeCourante() - annéeDeNaissance ;
  }
}

class Automobile code_reuses Personne (
     // Réutilisation de "annéeDeNaissance"
     // avec renommage en "annéePremièreImmatriculation"
     annéeDeNaissance -> annéePremièreImmatriculation,
     // Réutilisation de "âge()" sans renommage
     âge()) {
  Couleur couleur ;
}
```

Dans un esprit pragmatique (*Il existe des moments où le programmeur veut dupliquer un bout de code !*), nous proposons la création d'une relation de Réutilisation de code dont voici les valeurs des paramètres :

Name "Code_Reuse".

Kind import.

Context library.

[3] Dans un souci de simplicité, nous utilisons ici une syntaxe *à la Java enrichi du mot-clé code_reuses.*

Services <>.

Cardinality Comme la réutilisation de code n'offre pas de capacité polymorphique (nous le verrons plus loin), il n'y a aucun désavantage, au contraire, à la rendre multiple. La valeur de Cardinality est donc : $1 - \infty$.

Repetition A priori, il peut sembler que la répétition soit inutile dans ce contexte de réutilisation de code. Elle peut cependant s'avérer utile lorsque nous souhaitons non seulement réutiliser du code de la description-cible, mais également le dupliquer dans la description-source. La valeur allowed est donc pertinente ici pour les descriptions-cibles et cette valeur est ignorée pour les descriptions-sources. Nous avons donc <*ignoré*, allowed>.

Circularity Ici, rien n'empêche que deux descriptions réutilisent mutuellement (directement ou indirectement) des primitives. La valeur de Circularity est donc allowed puisque la relation Réutilisation de code admet la circularité. Cependant, il faut prendre garde à ce que cette circularité ne se retrouve pas au niveau d'une primitive, sans quoi nous serions dans un cas où le code réutilisé n'existe effectivement nulle part.

Symmetry false.

Opposite none.

Direct_access La réutilisation de code est, du point de vue du programmeur, une importation avec adaptation. Cependant, le code initial n'est, dans les faits, pas forcément recopié. Dans ce contexte, le but sémantique est bien de réutiliser dans la description-source du code de la description-cible. Ce dernier doit donc être accessible, la valeur de Direct_access est allowed.

Indirect_access Par contre, hormis pour le code commun, la source et la(es) cible(s) n'ont aucun autre rapport. Il n'y a donc pas de raison de permettre un accès indirect de la première vers la(es) seconde(s). La valeur choisie est donc forbidden.

Polymorphism_implication La réutilisation de code n'implique aucun polymorphisme, c'est une de ses spécificités. Nous sélectionnons donc la valeur none.

Polymorphism_policy Polymorphism_policy est ignoré puisque le paramètre Polymorphism_direction est à none.

Feature_variance Nous trouvons ici un cas intéressant pour le choix de la variance associée aux primitives. Nous pourrions penser que permettre une variance n'est pas pertinent puisqu'il s'agit d'une simple réutilisation de code. Cependant, voyons les choses d'un autre point de vue. Le code n'est pas modifié (cf. ci-dessous le paramètre Redefining qui interdit la redéfinition du corps des primitives). Donc rien n'empêche d'appliquer une règle de covariance aux paramètres de méthode et attributs. Sur le principe que l'objet spécialisé sait faire tout ce que peut réaliser l'objet plus général, le code reste en effet forcément valide. Pour les résultats de fonction, le même genre de raisonnement nous amène à autoriser la contravariance. Ajoutons que nous pensons que la non-variance sera finalement appliquée dans la plupart des cas. Mais pourquoi empêcher une modification valide ? La valeur choisie est donc <covariant, contravariant, covariant>.

Assertion_variance Ici notre choix relève d'un autre raisonnement. Les assertions sont généralement exprimées en fonction des primitives de la description courante. Or toutes les primitives de la description-cible n'ont pas forcément été

importées. De plus, nous agissons ici au niveau du code et non de sa sémantique. Quant aux invariants, ils n'ont à l'évidence aucun rapport. Nous proposons donc de ne pas contraindre les assertions de la description source par rapport à celles de la description cible. La valeur d'Assertion_variance est donc <non_applicable, non_applicable, non_applicable>.

Adding Aucune contrainte d'ajout n'a à s'appliquer dans le cadre d'une réutilisation de code. Adding prend donc la valeur allowed.

Removing Il est bien évident que nous ne souhaitons pas obliger à importer toutes les primitives de la description cible. C'est donc allowed qui est sélectionné ici.

Renaming Le renommage de la primitive réutilisée est une fonctionnalité particulièrement utile dans ce contexte. La valeur allowed est donc appropriée.

Redefining Conformément aux paramètres Feature_variance et Assertion_variance, les qualifieurs peuvent être redéfinis librement et la signature sous certaines contraintes. La redéfinition du corps est par contre non souhaitable. Par contre, la redéfinition des assertions est obligatoire puisque les assertions de la cible n'ont aucun sens dans la source. Redefining prend donc la valeur <mandatory, allowed, forbidden, allowed>.

Masking La réutilisation de code ne s'embarrasse pas de possibilités de masquage, démasquage, abstraction et concrétisation. La valeur forbidden est donc valide ici et dans les paramètres suivants.

Showing forbidden.

Abstracting forbidden.

Effecting forbidden.

Nous venons de décrire une relation de réutilisation de code et passons maintenant à une relation de vue.

Vue

Décrivons encore un nouveau type de relation entre classes. L'idée est ici de permettre la gestion d'une Vue d'un objet (avec un sens se rapprochant de celui que l'on trouve dans les systèmes de gestion de bases de données pour mieux présenter, mieux adapter ou mieux sécuriser des informations), c'est-à-dire offrir plusieurs représentations différentes du même objet. Il s'agit là encore d'un des usages (un peu détourné) de l'héritage que nous suggérons de mieux contrôler tout en le rendant plus explicite. Les paramètres de concept-relation peuvent prendre, pour cette relation, les valeurs suivantes :

Name "View".

Kind import.

Context library.

Services <>.

Cardinality Nous choisissons de modéliser une relation simple avec la cardinalité égale à $1-1$ car nous voulons une relation facile à mettre en œuvre. Une relation multiple (et rien ne nous interdit de faire plus tard "Multiple_View") permettrait de simuler les jointures mais devrait en contrepartie gérer la composition des descriptions-cibles.

Repetition valeur ignorée.

Circularity Nous souhaitons empêcher la circularité : une description ne peut pas être une vue d'une de ses vues. La valeur de Circularity est donc forbidden.

Symmetry Nous estimons que cette relation de Vue est symétrique. Construire une vue, c'est principalement supprimer ou ajouter des primitives ce qui est bien une opération symétrique. La valeur choisie est donc true. Remarquons que cela ne crée par un cycle dans le schéma de la relation. Une relation de Vue est bien posée entre une description-source (la vue) et une description-cible et pas l'inverse. Tout ce que nous voulons signaler, c'est que nous pourrions dire *est-une-vue-de* dans les deux sens sans nuire à la modélisation.

Opposite La relation de Vue ayant été déclarée comme symétrique, elle est automatiquement égale à son opposée. La valeur de Opposite est donc Vue.

Direct_access Il est inutile de dupliquer dans la description-source (la vue) ce qui est déjà accessible sous une forme adéquate dans la description-cible. Donc, nous souhaitons rendre accessible directement toutes les primitives de la description-cible dans la vue, sauf celles qui seront supprimées, masquées, etc. dans une clause d'adaptation. La valeur de Direct_access est allowed.

Indirect_access Une des qualités d'une vue est de masquer la forme réelle de la description-cible. Dans cet esprit, nous ne souhaitons pas permettre d'accès indirect (avec citation explicite de la description-cible). La valeur choisie est donc forbidden.

Polymorphism_implication Soit D une description quelconque et VD une description-vue de D. Il serait particulièrement intéressant que toute instance de D soit également instance de VD et vice-versa. Ainsi tout objet peut être considéré selon le point de vue de sa description (ici D) ou d'une vue (telle VD). Pour réaliser cela, nous choisissons la valeur both qui décrit un polymorphisme bi-directionnel. Ainsi l'ensemble des instances (directes et indirectes) d'une description et de chacune de ses vues sont similaires.

Polymorphism_policy La valeur de Polymorphism_policy n'est applicable que lorsque nous permettons la redéfinition de la signature des primitives, ce qui n'est pas le cas dans cette relation de Vue.

Feature_variance Notre choix est de considérer que la vue doit changer la liste des primitives accessibles mais pas modifier celles-ci. Nous verrons dans le paramètre Redefining que nous interdisons la modification de la signature des primitives. La valeur du paramètre Feature_variance sera donc en fait de toute façon ignorée.

Assertion_variance Pour la même raison que dans le paramètre Feature_variance, nous aurions pu vouloir ici la valeur <unchanged, unchanged, unchanged>. Cependant un problème peut alors apparaître. En effet, si nous retirons une primitive de la description-cible pour créer la vue, il est possible qu'une assertion, dans la vue, fasse appel à cette primitive et ne soit donc plus valide. Aussi notre choix est de laisser la liberté et la responsabilité au programmeur de modifier éventuellement les assertions pour pallier ce problème de cohérence. La valeur d'Assertion_variance est donc ici <non_applicable, non_applicable, non_applicable>.

Adding Une vue peut ajouter des primitives. Adding prend donc la valeur allowed.

Removing Plus souvent encore que l'ajout, la suppression de primitives de la description-cible nous semble un objectif important. C'est donc allowed qui est sélectionné ici.

Renaming Renommer une primitive peut s'avérer particulièrement utile dans une vue, même si on ne peut modifier son comportement. La valeur allowed est donc appropriée.

Redefining Spécificité de notre relation de Vue, la redéfinition des qualifieurs et signatures de primitive et des corps de méthode est interdite. Pour la raison expliquée ci-dessus dans le paramètre Assertion_variance, les assertions peuvent être redéfinies. Redefining prend donc la valeur <allowed, forbidden, forbidden, forbidden>.

Masking Le masquage peut s'avérer pratique lorsque le programmeur réalise une vue d'une vue. La première à être créée masque une primitive de sa description-cible et la vue de la vue la fait réapparaître... La valeur allowed est donc valide ici et dans le paramètre suivant.

Showing allowed.

Abstracting Il est interdit de redéfinir le corps des primitives. Dans cette optique, pouvoir rendre une primitive abstraite n'est pas applicable. La valeur de ce paramètre, et donc celle du suivant, est forbidden.

Effecting forbidden.

La figure 7.2[4] donne une exemple d'usage de la relation de Vue que nous venons de créer.

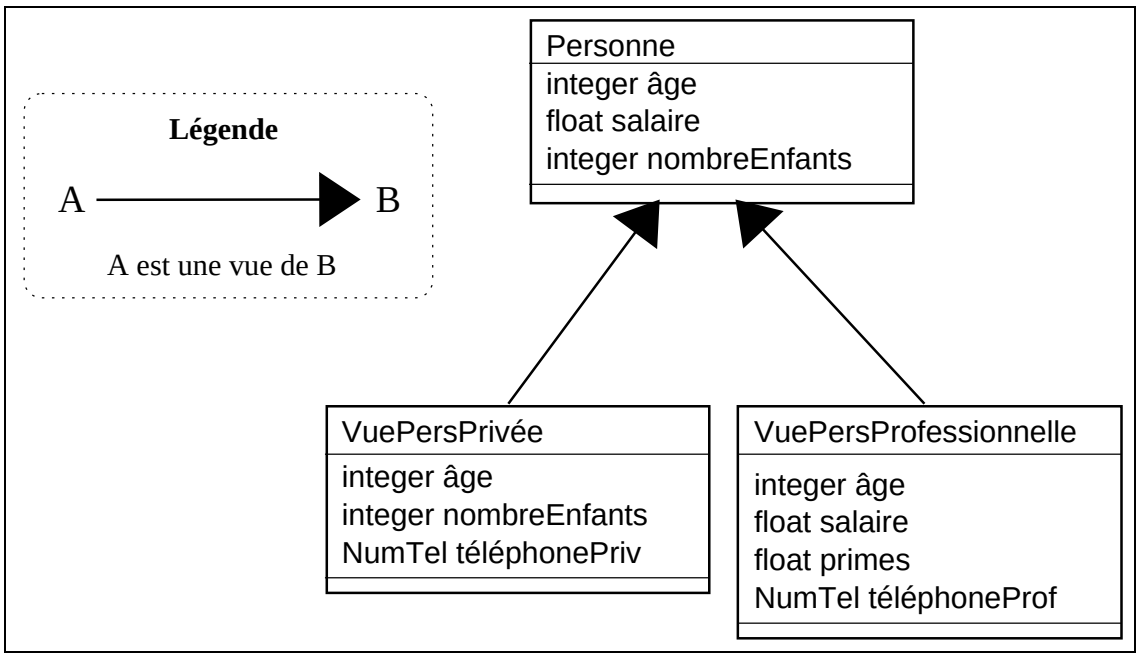

FIG. 7.2 – Exemple d'utilisation d'une relation de vue

Nous aimerions ajouter une remarque. Nous pouvons dire de cette relation :

[4]Rappelons que la légende des figures pour tout ce document est donnée dans la figure 3.1 page 48.

– qu'elle possède un polymorphisme bi-directionnel, ce qui a pour conséquence de rendre équivalents les types décrits par la description-source et la description-cible et
– que des primitives peuvent exister dans la description-source sans être définies dans la description-cible, et vice-versa.

Ces faits ont pour effet de ne pas assurer de toujours trouver, lors de la liaison dynamique, une primitive adéquate. Par exemple si nous créons un objet container pour y conserver des Personnes. Ce container pourra, sans aucun souci de typage, recevoir également des VuePersPrivées et des VuePersProfessionnelles. Si nous appliquons alors la primitive salaire à tous les membres du container, les VuePersPrivées ne sauront pas y répondre. Dans ce cas, les réactions classiques des langages à objets peuvent s'appliquer : typage explicite (type casting) vers Personne par le programmeur, avertissement à la compilation, exception à l'exécution, ...

Bilan et tableaux comparatifs

Les sections précédentes ont eu pour but de montrer quel doit être le travail du méta-programmeur utilisant le modèle *OFL*. Il s'agit évidemment du cas optimiste, celui dans lequel les actions donnent la sémantique attendue par le méta-programmeur aux valeurs choisies pour les paramètres. Notre but est de rendre ces actions les plus complètes possibles, et de les compléter au fur et à mesure des besoins, pour que cet optimisme s'applique dans la majorité des cas. Si le méta-programmeur souhaite modifier ou améliorer les actions, il retombe alors dans une cadre habituel de méta-programmation pour lequel il doit lui-même compléter ou fournir les algorithmes génériques reflétant la sémantique opérationnelle des langages.

Précisons qu'il est de la responsabilité du méta-programmeur de choisir un ensemble de valeurs cohérent pour les paramètres. *OFL* est, en ce sens, bien un modèle et non une méthodologie. Il offre les bases et les outils de modélisation d'un langage, les moyens de le mettre en œuvre, mais pas la liste de toutes les sémantiques possibles. Ajouter un méthodologie qui assiste le méta-programmeur et lui indique les voies à suivre et les pièges les plus fréquents est une des perspectives de ce travail de thèse (cf. section 8.1 page 175).

Enfin, nous aimerions terminer en fournissant un tableau récapitulatif (tableau 4.2 page 84) des cinq composants-relations que nous venons de décrire. Vous pouvez y voir que les différences entre ces composants sont relativement faibles par rapport aux apports que nous pouvons en attendre. Cette situation est tout à fait normal puisque toutes ces relations sont plus ou moins bien implémentées grâce au seul mécanisme d'héritage (parfois aidé de l'agrégation) dans les langages traditionnels.

Remarquez que dans le tableau 4.2 page 84, seuls les paramètres pertinents pour les composants-relations d'importation ont été signalés, tous nos exemples étant choisis dans cet ensemble.

7.2 Apports du modèle

Cette section décrit les apports de l'usage du modèle *OFL*. L'objectif initial de notre étude était l'ajout de relations d'importation dans un langage à objets, pour

Paramètre	Spécialisation	Implémentation	Généralisation	Réutilisation de code	Vue
Name	"Specialisation"	"Implementation"	"Generalisation"	"Code_Reuse"	"View"
Kind	import	import	import	import	import
Context	library	library	library	library	library
Services	<>	<>	<>	<>	<>
Cardinality	$1-1$	$1-\infty$	$1-1$	$1-\infty$	$1-1$
Repetition	*ignoré*	<*ignoré*, forbidden>	*ignoré*	<*ignoré*, allowed>	*ignoré*
Circularity	forbidden	forbidden	forbidden	allowed	forbidden
Symmetry	false	false	false	false	true
Opposite	Généralisation	none	Spécialisation	none	Vue
Direct_access	allowed	forbidden	allowed	allowed	allowed
Indirect_access	allowed	forbidden	allowed	forbidden	forbidden
Polymorphism_-implication	up	up	down	none	both
Polymorphism_-policy	<overriding, overriding>	<overriding, overriding>	<overriding, overriding>	*ignoré*	*ignoré*
Feature_variance	<covariant, nonvariant, covariant>	<covariant, nonvariant, covariant>	<contravariant, nonvariant, contravariant>	<covariant, contravariant, covariant>	*ignoré*
Assertion_-variance	<strengthened, weakened, strengthened>	<strengthened, weakened, strengthened>	<weakened, strengthened, weakened>	<non_applicable, non_applicable, non_applicable>	<non_applicable, non_applicable, non_applicable>
Adding	allowed	allowed	forbidden	allowed	allowed
Removing	forbidden	forbidden	allowed	allowed	allowed
Renaming	allowed	allowed	allowed	allowed	allowed
Redefining	<allowed, allowed, allowed, allowed>	<allowed, allowed, forbidden, allowed>	<allowed, allowed, allowed, allowed>	<mandatory, allowed, forbidden, allowed>	<allowed, forbidden, forbidden, forbidden>
Masking	forbidden	forbidden	allowed	forbidden	allowed
Showing	allowed	allowed	forbidden	forbidden	allowed
Abstracting	forbidden	forbidden	allowed	forbidden	forbidden
Effecting	allowed	mandatory	forbidden	forbidden	forbidden

TAB. 7.1 – Cinq composants-relations

enrichir les mécanismes d'héritage.

En se généralisant et en s'étendant, le modèle *OFL* offre aujourd'hui de nombreux avantages. En voici certains :

Flexibilité C'est le premier des apports, il fait même partie du nom du modèle. Il rejoint une volonté des concepteurs de *CLOS* [68, 70] ou *Smalltalk* [58, 57]. La flexibilité offerte par *OFL*, c'est la possibilité d'adapter son langage de programmation à ses besoins en :
- modifiant par hyper-généricité sa sémantique opérationnelle (par le changement de la valeur des paramètres des *OFL*-composants),
- modifiant par méta-programmation sa sémantique opérationnelle (par le changement du code des actions des *OFL*-composants) et/ou
- ajoutant ou supprimant des sortes de description ou relation (des *OFL*-composants) utiles dans un contexte donné.

Documentation Il s'agit d'un objectif mésestimé mais qui nous tient à cœur. Cet apport se place à deux niveaux :

1. Le fait d'avoir établi un liste de paramètres décrivant la sémantique opérationnelle des descriptions et relations d'un langage améliore notablement sa documentation. Le programmeur peut, en effet, soit se référer à un traditionnel manuel de référence, soit analyser son langage en terme de valeur des paramètres d'*OFL*-composants voire, avec une granularité encore plus fine, analyser le code des actions d'*OFL*-composants. Soyons bien précis : nous n'avons pas l'ambition de remplacer un manuel de référence, qui reste toujours indispensable, mais bien d'enrichir la documentation par de nouvelles informations. Il s'agit ni plus ni moins d'offrir le code *ouvert* du traducteur (interprète ou compilateur) du langage.

2. De plus, l'existence de descriptions et relations plus précises, plus spécifiques que l'agrégation et l'héritage peut améliorer sensiblement la documentation du code de l'application générée. En effet, l'usage d'une spécialisation, d'une généralisation, d'une vue, etc. est beaucoup plus explicite que celui d'un héritage. Il est ainsi possible de générer un documentation automatique bien plus précise et pertinente.

Lisibilité Ici, nous nous plaçons aux niveaux des bénéfices du langage décrit. Plus le code est précis, moins il contient de non-dit, plus il est lisible. Ainsi, l'avancée de *Java* [55, 61, 3, 39] en termes de descriptions et relations (avec l'apparition de interface et implements notamment) va dans ce même sens : un lecteur qui lit implements ou extends au lieu d'un toujours identique inherits sera infiniment plus renseigné. Nous souhaitons encore pousser cette idée en offrant la capacité des créer de nouveaux composants de langage affectés à une tâche spécifique à un domaine d'application ou au moins, plus précise que ce que nous trouvons actuellement.

Maintenabilité C'est un des apports les plus importants. Une des grandes difficultés actuelles de la programmation est constituée par les problèmes de maintenance. En effet, il est extrêmement complexe de reprendre le code d'un programme déjà écrit par une autre personne ou une autre équipe dans un autre temps. Les apports précédents (documentation et lisibilité) aident de manière importante le mainteneur à s'y retrouver dans ce qui se trouve souvent être une jungle en lieu et place d'un programme.

Contrôle Plus une description ou une relation est précise, plus les contrôles qu'il est possible de réaliser sont étendus. Ainsi une relation de *réutilisation de code* (cf. section 7.1.2 page 160) interdit le polymorphisme, et cela peut être vérifié, là où l'indication d'un héritage traditionnel n'aurait permis aucun contrôle. Les composants logiciels (descriptions d'une bibliothèque par exemple) offrent donc une robustesse et une fiabilité accrues par une meilleure adéquation à la sémantique qu'a souhaité leur donner leur programmeur. Ce point est particulièrement important. Le contrôle automatique le plus étendu possible est à notre avis un gage très important de la qualité d'un code source. Si nous étendons l'expressivité d'un langage en déployant en plus de tels contrôles, cet objectif est atteint.

Performance Plus les indications du programmeur sont précises, plus les processus d'optimisation des performances de l'interprète ou du compilateur sont efficaces. Il ne s'agit pas ici de tomber dans l'excès du register de *C* [69, 49] ou du virtual de *C++* [91, 92] qui indiquent comment optimiser. Il s'agit plutôt de profiter, au niveau de l'interprète/compilateur, des nouvelles informations, plus précises, données par le programmeur. L'apport de performance est un des plus difficiles à atteindre. Il repose en effet souvent sur l'expérience et une longue maturité des outils. Il est encore compensé, au sein du projet *OFL*, par la jeunesse de ce dernier.

Évolution *OFL* favorise l'évolution. Ici aussi, deux niveaux d'apport peuvent être dégagés :

1. Les capacités d'évolution des applications sont améliorées pour les mêmes raisons qui accroissent la maintenabilité : meilleures documentation et lisibilité. De plus, l'application peut profiter des modifications et améliorations apportées à son langage (cf. point suivant).

2. Les langages eux-mêmes bénéficient de notre approche. *OFL* décrit ces langages. Il permet de les adapter à des besoins spécifiques, de les étendre, de les faire évoluer. Le méta-programmeur voit sa tâche grandement facilitée par l'hyper-généricité (système de paramétrage) qui lui évite souvent d'écrire du code, ce dernier étant complexe et délicat. Il a donc la possibilité de tester directement les améliorations qu'il compte intégrer à son langage, d'exécuter un nouveau comportement de son langage sur une application existante et de valider ainsi son approche.

Adéquation Cet apport, parmi les plus importants, tend à rapprocher les méthodes de conception telles *UML* [5, 90, 83] des langages de programmation comme *Java* [55, 61, 3, 39]. Lors de la phase de conception, il est fréquent de définir des relations inter-descriptions plus diversifiées que celles dont disposeront par la suite les programmeurs. C'est même l'intérêt du concepteur de décrire des liaisons à sémantique précise. *OFL* permet de décrire de nouvelles relations, au sein du langage de programmation, pour éviter de se trouver trop éloigné de celles définies lors de la phase de conception.

Enfin, de manière générale, nous pouvons dire que l'apport d'*OFL* tient principalement à l'adaptabilité qu'il intègre au sein des langages, tout en offrant des dispositifs de contrôle (typage, assertions, ...) dignes du génie logiciel, et à la présence de nouvelles informations pertinentes aussi bien pour le programmeur que pour l'interprète/compilateur.

7.3 Originalités par rapport aux autres approches

Le modèle *OFL*, tel que nous l'avons conçu et présenté, dispose de plusieurs originalités par rapport aux approches *méta* connues telles *CLOS-MOP*, *Smalltalk*, ... Nous rappelons ici ces différences.

La première originalité d'*OFL* est de ne pas reposer sur un langage de programmation donné. Nous avons, depuis le début, voulu éviter l'écueil qui consiste à partir d'un seul langage comme base. Nous espérons ainsi avoir développé une approche plus générale et mieux contribué à la rendre applicable à de nombreux langages.

Originalité plus importante, notre volonté de placer la relation inter-description au centre du modèle plutôt que la description elle-même, comme c'est le cas dans les autres systèmes méta. Nous pensons que la structure de description est aujourd'hui assez bien maîtrisée et que l'effort doit être concentré sur la manière d'utiliser, de réutiliser, d'importer, de mettre à profit ces descriptions.

Enfin, nous avons souhaité simplifier, autant que possible, le travail du métaprogrammeur et, c'est dans cet esprit que nous avons défini un système de paramètres. Sans résoudre, bien évidemment, l'intégralité des problèmes de métaprogrammation, ce système nous paraît être une approche valide pour libérer le métaprogrammeur des cas les plus standards et lui offrir l'opportunité de se concentrer sur la partie originale du langage qu'il décrit. L'ensemble des actions, véritables points d'entrée dans un compilateur, une interprète ou une machine virtuelle, constitue également un apport non négligeable dans ce sens.

7.4 Outils logiciels autour d'*OFL*

OFL est un modèle que nous avons décrit. Une fois cette description bien avancée, nous nous sommes occupés d'entamer la création d'outils logiciels qui mettent en œuvre les principes et entités d'*OFL* et permettent d'en tirer parti. Nous présentons ainsi ci-dessous les quatre principaux outils en cours de développement.

7.4.1 *OFL-DB*

Tous les outils qui gravitent autour d'*OFL* doivent partager des données. Ces données sont par exemple les *OFL*-concepts et les *OFL*-atomes. Mais n'oublions pas également celles qui seront les plus importantes pour les programmeurs, les bibliothèques d'*OFL*-composants. Nous avons donc besoin de conserver ces entités dans un formalisme standard, pour permettre un partage aisée, et durable, pour pouvoir envisager leur réutilisation.

La version actuellement en développement d'*OFL-DB* repose sur un système de gestion de bases de données orienté objets nommé *POET*[5] [88]. Toutes les entités *OFL* sont ainsi sauvegardées sous forme d'objets et, plus important, disponibles au travers d'une API [6] *Java* pour les autres outils *OFL*. *POET* respecte également les spécifications de l'*ODMG* [20] et est capable d'exporter ses données sous un format *XML* [80, 102], capacité dont nous ne tirons pas encore parti. Le choix de *POET* reste cependant anecdotique en regard des bases *ODMG*, *XML* et *Java* choisies pour leur propriété de standards.

[5] *POET* a été récemment renommé en *FastObjects*.

[6] *Application Programming Interface*

OFL-DB constitue ainsi l'interface au socle de données sur lequel reposent les autres outils *OFL*. Il fournit la réification persistante du modèle et de l'ensemble des bibliothèques d'*OFL*-composants créées par les méta-programmeurs.

7.4.2 *OFL-Meta*

OFL-Meta est le second outil en cours de développement. Il est destiné aux méta-programmeurs. Son rôle est de permettre et de faciliter la création de nouveaux *OFL*-composants.

Le but d'*OFL-Meta* est d'offrir au méta-programmeur la capacité de décrire un langage. Pour cela, nous proposons une structure d'arbre qui présente une vision synthétique du modèle *OFL* appliqué à un langage avec ses *OFL*-concepts et *OFL*-composants. Au départ, l'ensemble des *OFL*-composants est vide à l'exception du composant-langage qui représente le langage en cours de réalisation.

L'interface permet également d'importer des bibliothèques d'*OFL*-composants réutilisables (Context=library) ; ceux-ci viennent se placer dans un sous-arbre spécifique et indépendant de celui du langage à construire.

Le méta-programmeur est alors devant une alternative pour créer les *OFL*-composants qui constitueront son langage :

- Soit il crée de toutes pièces un nouvel *OFL*-composant (composant-description ou composant-relation) en instanciant un *OFL*-concept[7]. Dans ce cas, tous les éléments de réification et paramètres de cet *OFL*-composant sont placés à leur valeur par défaut. Le méta-programmeur doit simplement consulter chaque valeur pour vérifier qu'elle convient à son usage ou pour la modifier.
- Soit il importe dans le langage courant un *OFL*-composant (composant-description ou composant-relation) présent dans une bibliothèque, voire dans un autre composant-langage. Il a alors le choix entre une réutilisation quasiment *telle quelle*. Mais il peut aussi modifier l'*OFL*-composant importé pour l'adapter à ses besoins.

Lorsqu'il a terminé de créer son langage, il lui faut le sauvegarder pour le mettre à la disposition des programmeurs. La sauvegarde est actuellement réalisée vers des fichiers textes disposant d'une syntaxe spécifique et simple. L'intégration vers *OFL-DB* est en cours de réalisation. De plus, une exportation au format *XML* (soit directement, soit par utilisation d'*OFL-DB*) fait partie des projets à courts termes pour permettre une diffusion accrue et facilitée des *OFL*-composants.

La figure 7.3 page ci-contre décrit schématiquement l'interaction entre *OFL-Meta* et *OFL-DB*.

7.4.3 *OFL-ML*

Les entités de base du modèle sont toutes conservées par *OFL-DB*. Le méta-programmeur est intervenu et a décrit un langage, en fait la sémantique opérationnelle de ses descriptions et relations au travers des paramètres, à l'aide d'*OFL-Meta*. Il laisse maintenant la place au programmeur.

Ici, l'outil requis est *OFL-ML*. Son nom est dérivé de celui d'*UML* car il emprunte à ce dernier une partie de son formalisme graphique [87]. *OFL-ML* permet au programmeur, en utilisant les composants décrits dans *OFL-Meta*, de créer graphique-

[7] À ce stade, il n'est pas possible de créer un nouvel *OFL*-concept. Cela devrait être faisable dans l'avenir grâce à un système de *bootstrap* pour *OFL*, en cours de conception.

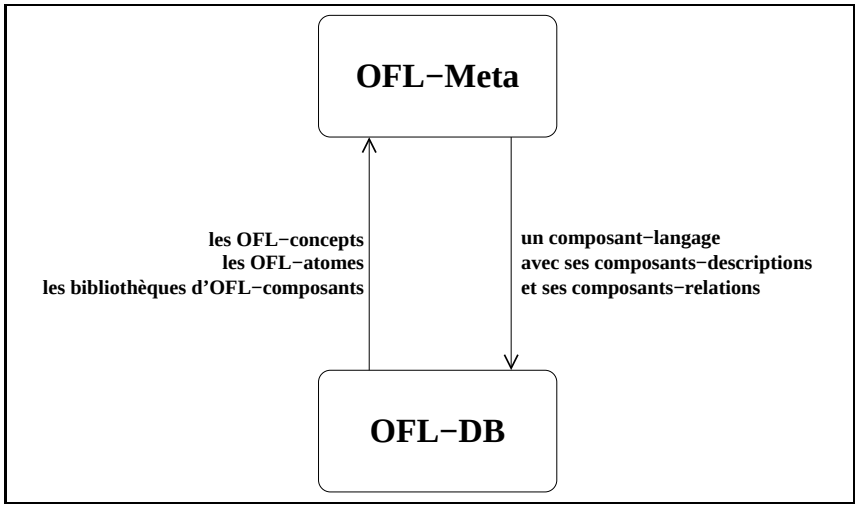

FIG. 7.3 – *OFL-Meta* et *OFL-DB*

ment des descriptions, de décrire les relations entre elles et de donner un corps aux méthodes, initialiseurs, ...

OFL-ML importe tout d'abord la définition d'un langage décrit en *OFL*, c'est-à-dire la définition d'un composant-langage et des tous ses composants-descriptions et composants-relations. Ceux-ci sont alors mis à la disposition du programmeur qui les utilisent pour dessiner le graphe de descriptions de son application. De plus, il a la possibilité d'ouvrir une interface de saisie textuelle pour associer des instructions, partout où cela est utile et notamment dans le corps des méthodes. L'outil suivant, *OFL-Parser*, produisant du *Java*, c'est également du *Java* que le programmeur utilise dans la version actuelle d'*OFL-ML*. Ces bouts de code ne sont pas interprétés par *OFL-ML* et ils seront intégrés tels quels dans le code produit par *OFL-Parser*.

Nous présentons dans les figures 7.4 page suivante et 3.2.2 page 53 deux copies de fenêtres d'*OFL-ML*. La figure 7.6 page 173 propose quant à elle de montrer les interactions entre *OFL-ML* et *OFL-DB*.

7.4.4 *OFL-Parser*

Le méta-programmeur puis le programmeur ayant terminé leur travail, il reste maintenant à traduire les différents éléments du modèle et la définition de l'application pour qu'elle devienne exécutable. *OFL-Parser* est chargé de ce travail. Si son interface peut s'avérer simpliste (les interactions avec l'utilisateur sont particulièrement limitées dans cette phase), ce n'est pas le cas de sa tâche.

En effet, le but d'*OFL-Parser* est de compiler puis d'exécuter l'application décrite en *OFL-ML*. Pour cela, il doit mettre en œuvre les actions d'*OFL* qui renferment le code qui tient compte des paramètres des composants. *OFL-Parser* est sans doute l'outil qui nous demandera le plus d'attention. En effet, le code exact de toutes les actions n'est pas encore déterminé et celui-ci, générique et devant prendre en compte un grand nombre de combinaisons de valeurs de paramètre, est particulièrement complexe. Notre choix s'est porté sur *Java* comme langage-cible. Ainsi, le code écrit par le programmeur avec *OFL-ML* peut-il être mis à profit sans ennui.

En d'autres termes, *OFL-Parser* a pour objectif de composer le code des actions *OFL* avec la structure et le code de l'application décrite afin de produire du *Java* qui

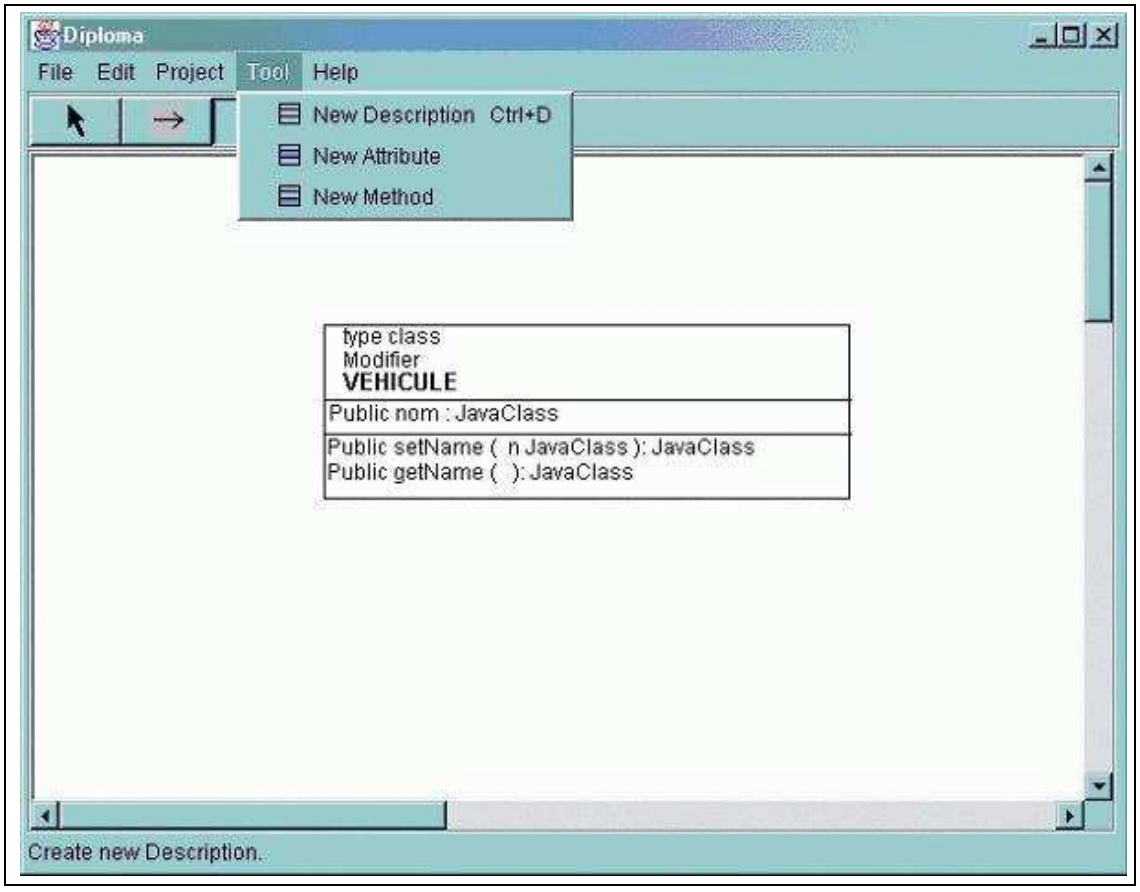

FIG. 7.4 – Un exemple d'une fenêtre d'*OFL-ML*

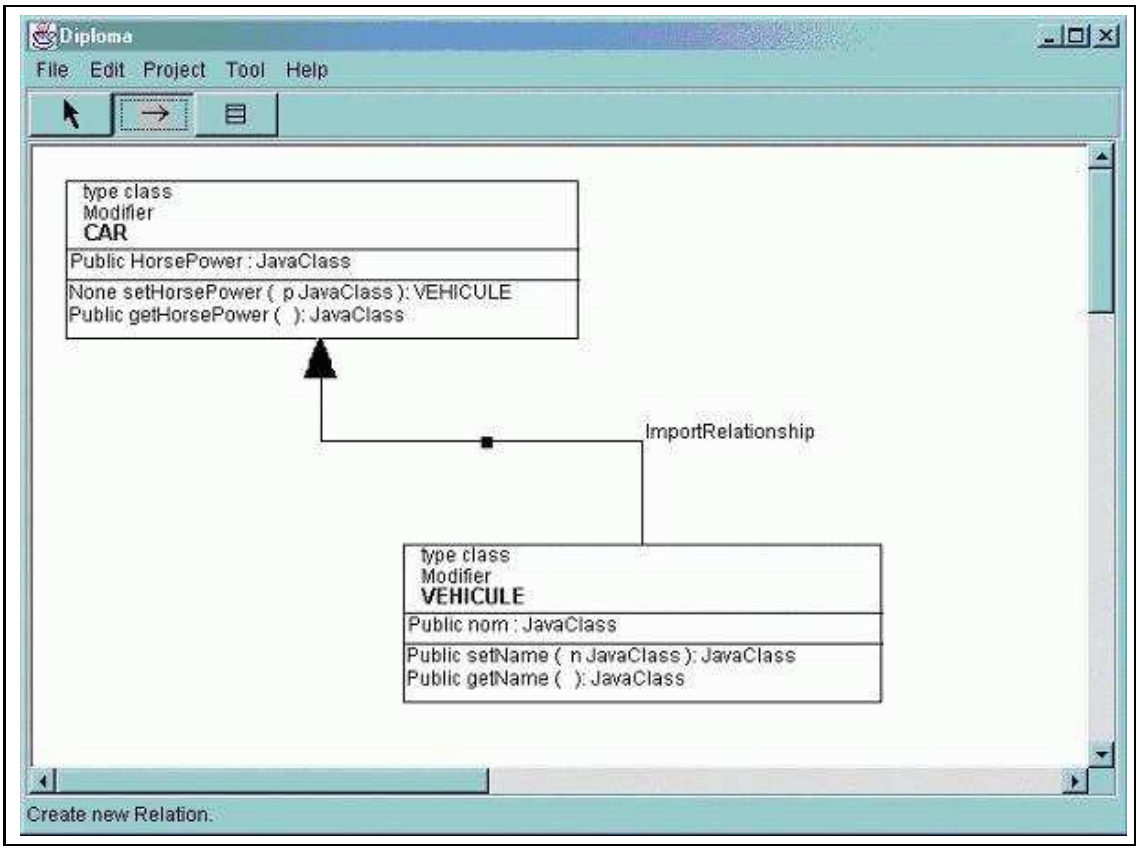

FIG. 7.5 – Un autre exemple d'une fenêtre d'*OFL-ML*

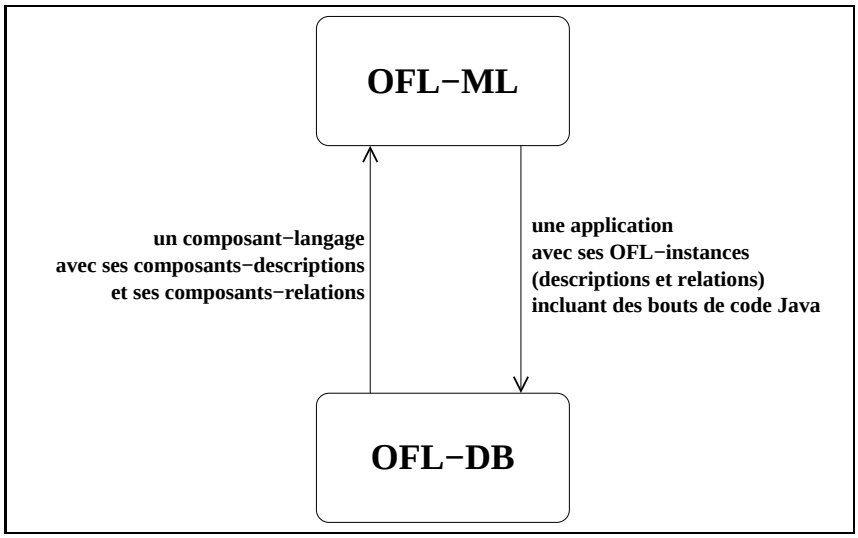

FIG. 7.6 – *OFL-ML* et *OFL-DB*

sera ensuite compilé ou interprété par un outil standard.

La réalisation d'*OFL-Parser* a commencé mais est encore assez loin d'être achevée. La conclusion de ce travail, plus encore que celui mettant en place les autres outils *OFL*, constituera une validation importante de la pertinence du modèle.

Les interactions entre *OFL-Parser* et *OFL-DB* sont décrites sur la figure 7.7.

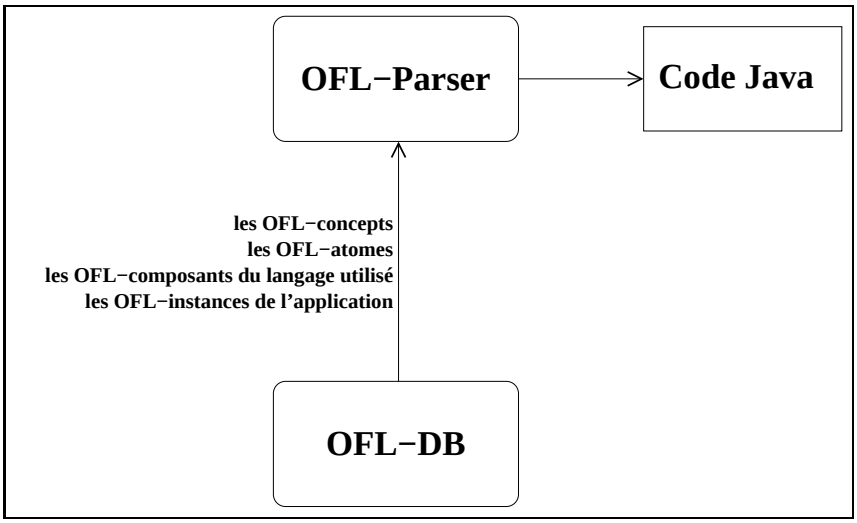

FIG. 7.7 – *OFL-Parser* et *OFL-DB*

7.4.5 Bilan

Nous venons de décrire les quatre principaux outils qui formeront un environnement de méta-programmation et de programmation associé au modèle *OFL*. Tous ces outils sont en cours de développement. Le travail restant pour parvenir à une première version de cet environnement est important. La programmation des algorithmes des actions en constitue l'aspect sans aucun doute le plus délicat et probablement le plus long.

La figure 7.8 résume et rappelle l'usage de chaque outil placé dans sa phase d'utilisation.

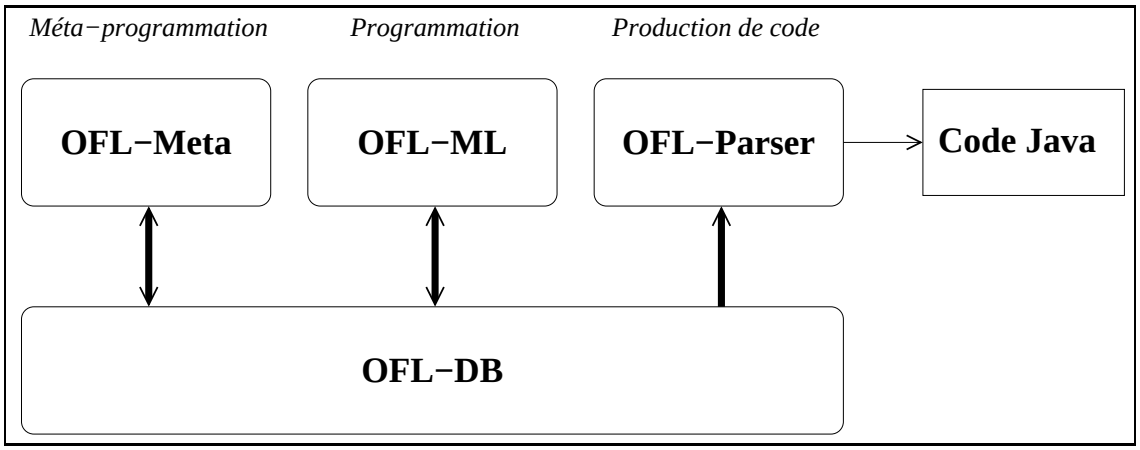

FIG. 7.8 – Les outils *OFL* dans leur ensemble

Chapitre 8

Perspectives et conclusions

Ce dernier chapitre présente dans un premier temps les perspectives probables et possibles de poursuites de ce travail de thèse. Il conclue ensuite ce document.

8.1 Perspectives

Comme souvent dans un projet de recherche issu d'un travail de plusieurs années, les perspectives sont nombreuses. Nous présentons ici celles qui nous semblent les plus essentielles. Nous entamerons probablement les trois premières et la dernière durant l'année universitaire suivant la soutenance de cette thèse, dans le cadre d'études post-doctorales. Les deux autres font l'objet de deux nouvelles thèses.

Poursuite de l'implantation Notre première volonté naturelle pour les travaux futurs concernant *OFL* est de poursuivre la tâche d'implémentation qui a débuté. L'avancement de cette partie à réaliser est en effet d'une grande importance car elle validera notre approche en même temps qu'elle la concrétisera. Les deux outils les plus significatifs pour l'utilisateur sont *OFL-Meta* et *OFL-ML*. Cependant, celui qui nous réserve le plus de travail est *OFL-Parser* pour la simple raison qu'il renfermera le code des actions qui forment la partie opérationnelle et originale d'*OFL*. Une autre capacité essentielle qui reste à programmer est la possibilité d'exporter et importer des définitions *XML* des composants du modèle.

Formalisation de la définition Parallèlement à cette implantation, nous pensons qu'*OFL* aurait tout à gagner à voir sa définition représentée à l'aide d'un formalisme prévu à cet effet tel *MOF*. La définition que nous en avons donnée dans ce document repose principalement sur une description en langage naturel. Si ce choix permet d'assurer une certaine fluidité aux explications, il laisse par contre à désirer en ce qui concerne la précision et la complétude de la définition. *MOF*, et probablement *UML*, seront donc plus mis à profit dans les prochaines versions d'*OFL*.

Méthodologie et assistance Le modèle *OFL* donne un cadre pour réaliser des activités de méta-programmation en centrant sa démarche sur les relations entre classes. Il possède intrinsèquement un défaut en commun avec les autres protocoles méta-objets : tout repose sur les épaules du méta-programmeur. Or, l'activité de méta-programmation, de définition du comportement d'un langage ou d'un système, est particulièrement complexe. Notre démarche de définition de paramètres hyper-génériques va évidemment dans le sens d'une simplification

nette de cette tâche. Nous souhaitons, de plus, associer à *OFL* deux types de fonctionnalité :

1. Une **méthodologie** doit être définie. Elle décrira au méta-programmeur l'ensemble des tâches qu'il doit réaliser pour accomplir son œuvre. Par exemple, lorsqu'il veut modifier un élément de sémantique opérationnelle d'un langage déjà défini en *OFL*, où doit-il intervenir ? Lorsqu'il souhaite ajouter un concept-relation à un langage, comment faire ? Quels points contrôler pour garantir la faisabilité de sa démarche, pour vérifier la compatibilité de la nouvelle relation avec le reste du langage ? Cette méthodologie devrait prendre, à notre sens, la forme d'une documentation textuelle, peut-être une partie d'un futur *Guide du méta-programmeur OFL*.
2. Une **assistance** pourrait et devrait être associée à cette méthodologie. Ici, il ne s'agit plus d'une documentation mais d'une fonctionnalité de l'environnement de programmation d'*OFL* pour conseiller, guider et contrôler les réalisations du méta-programmeur. Nous pensons donc à une sorte de *wizard*, un composant logiciel actif, qui assisterait cette tâche complexe. Cette volonté est particulièrement importante, peut-être plus pour *OFL* que pour les autres protocoles méta-objets. En effet, *OFL* se destine de manière prononcée à des tâches industrielles avec un typage fort, des assertions contrôlant une partie de la sémantique opérationnelle, des outils logiciels centrés sur les aspects pratiques de la programmation, ... Nous souhaitons donc mettre en avant cette caractéristique d'assistance qui dans une grande mesure pourra mettre en œuvre les aspects méthodologiques présents dans la fonctionnalité décrite ci-dessus. Ajoutons enfin que ce dispositif d'assistance pourrait prendre plusieurs formes, de la simple activation de messages lors la compilation à la présence d'un personnage virtuel ou de fenêtres interactives tout au long de l'utilisation de l'environnement de programmation. Dans tous les cas, il devra être activable et désactivable très facilement pour ne pas entraver la liberté du méta-programmeur. Il s'agit donc bien d'une assistance et non d'une prise de contrôle.

Séparation des préoccupations dans le modèle Cette perspective précise fait l'objet d'une autre thèse de doctorat, débutée en 1999 et qui s'appuie sur le présent travail. Ce sujet est réalisé par Adeline Capouillez sous la direction de Philippe Lahire.

OFL offre la capacité de décrire un langage de programmation en lui associant un comportement spécifique et en encourageant la définition de relations entre classes précises. Cependant, un certain nombre de préoccupations sont communes à un ensemble de langages et pourraient être factorisées. Il en est ainsi des capacités de persistance, de distribution, de concurrence, d'introspection, ... L'une des technologies les plus prometteuses pour résoudre ce genre de problème est la séparation des préoccupations (*Separation of Concerns*), souvent mise en œuvre au travers de la programmation par aspects.

Le travail d'Adeline Capouillez consiste à intégrer au niveau du modèle une préoccupation. Ainsi, si le modèle décrit les langage L1 et L2, le service[1] Persistance pourra être appliqué à ces deux composants-langages pour créer L1-persistant et L2-persistant. Il faut donc définir ici ce qu'est un service et comment il peut être greffé à un langage existant, formé d'*OFL*-composants. Précisons qu'ici c'est le

[1] Un service est la concrétisation d'une préoccupation.

méta-programmeur qui dispose de capacités de séparation des préoccupations et non le programmeur.

Séparation des préoccupations dans le langage La précédente perspective s'occupe de séparation des comportements au niveau du modèle. Celle qui nous occupe ici se place au niveau du langage de programmation décrit par le modèle. Une troisième thèse de doctorat, réalisée par Laurent Quintian et dirigée comme la précédente par Philippe Lahire, débute en effet, en cette année 2001, pour étudier la manière d'insérer des capacités de séparation des préoccupations dans un langage et en faire donc profiter le programmeur. Ce travail consiste notamment à évaluer les alternatives à une approche par protocole méta-objet pour élargir notre champs de recherche, mais aussi à vérifier la capacité d'*OFL* à décrire un langage disposant d'une technique de séparation des préoccupations telle la programmation par aspects.

Amélioration du modèle Dernière perspective que nous évoquerons ici, l'amélioration du modèle n'est en fait pas un but séparé des autres. Comme toute réalisation humaine, *OFL* présente de nombreux inconvénients et reste très imparfait. Nous nous attacherons à corriger les imperfections et à développer la qualité de ce modèle dans le futur. Toutes les perspectives présentées ci-dessus ne manqueront pas de nous y aider. Il faudra notamment nous attacher à décrire avec précision des relations d'utilisation qui n'ont pas été étudiées en profondeur durant cette étude. Cela nous amènera peut-être, par exemple, à différencier un attribut d'une variable locale, d'un paramètre de méthode ou encore d'un résultat de fonction. Nous pensons qu'une réflexion portant sur les règles de visibilité et espaces de nommage serait également la bienvenue. Pour l'instant gérés par le système de *qualifieurs* au sein d'*OFL*, ils gagneraient sans aucun doute à être modélisés de manière plus précise.

8.2 Conclusions

OFL est un modèle, un protocole méta-objet, dont une spécificité est de ne pas s'attacher à un langage sous-jacent particulier. Il décrit et réifie, de manière générale, les composants essentiels des langages à objets à classes et affecte aux sortes de descriptions et de relations de ceux-ci des paramètres permettant de modifier, d'adapter leur comportement ; cela en limitant au maximum l'écriture d'un code forcément complexe.

L'architecture du modèle *OFL* est aujourd'hui réalisée, les paramètres hypergénériques ont été mis en évidence, les actions définies. Quatre outils logiciels sont actuellement en cours de développement : ils feront sans aucun doute encore évoluer le modèle *OFL* et permettront de valider notre approche de manière pragmatique.

Pour terminer ce manuscrit, nous avons choisi de changer complètement de points de vue et de lister selon un nouvel angle trois des spécificités du modèle *OFL* que nous avons décrit.

Les relations au cœur de l'implantation De la même manière que les langages de programmation ont peu à peu fait primer les comportements sur les états, nous pensons que les relations entre classes prennent une importance grandissantes face aux classes elles-mêmes. *Java* a ouvert ainsi prudemment la voie vers le grand public avec ses interfaces et ses relations d'implémentation. Nous croyons aussi que les récentes avancées en matière de séparation des préoccupa-

tions vont exactement dans ce sens-là en déstructurant, distribuant la structure de données au travers de nouvelles interactions et relations. *OFL* se veut être un outil ouvert dans cette perspective.

Un protocole méta-objet générique Les protocoles méta-objets que nous avons étudiés sont tous créés au-dessus d'un langage spécifique, souvent *C++* ou *Java*, même quand ils se veulent généralisables. Nous avons entamé une démarche inverse, plus proche de celle d'*UML* ou de *MOF*, qui est de décrire une généralisation des composants de langage. Plus difficile — il faut ensuite parvenir à définir un vrai langage — mais plus prometteuse à notre avis, cette approche nous a conduit à décrire ce que nous nommerions aujourd'hui un protocole méta-objet générique capable d'occuper une couche méta au-dessus de plusieurs langages : une avancée vers la difficile interopérabilité.

La séparation des préoccupations dans l'application Vous l'aurez compris, la séparation des préoccupations est devenue l'un de nos chevaux de bataille. *OFL* permet d'ores et déjà de l'appliquer, sous la direction du méta-programmeur. Celui-ci peut en effet décrire des structures de données (composants-descriptions) et des relations entre elles (composants-relations). Nous retrouvons donc ici l'idée qu'une application est une superposition de graphes de structures de données (l'un de ces graphes est celui d'héritage, un autre celui d'agrégation, un autre celui d'évolution, ... Chaque couche représente une préoccupation, chaque interaction entre couche est une composition décrivant l'interaction de deux préoccupations. L'idée est lancée, reste sans doute à la développer.

Bibliographie

[1] S. ABITBOUL, P. BUNEMAN, et D. SUCIU. *Data on the Web, from Relational to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[2] ARBORTEXT. « SGML : Getting Started - A Guide to SGML (Standard Generalized Markup Language) and Its Role in Information Management ». `http://www.arbortext.com/data/Getting_Started_with_SGML/-body_getting_started_with_sgml.html`, 1995.

[3] K. ARNOLD et J. GOSLING. *The Java Programming Language*. The Java Series... from the Source. Sun Microsystems, 3e édition, 2000. `http://java.sun.com/docs/books/javaprog/`.

[4] L. AYACHE, P. CRESCENZO, et J.-L. PAQUELIN. « From Software Engineering to Linguistic Engineering ». *1st International Workshop on Applications of Constraint-Based Programming to Computational Linguistics*, mai 1998. `http://www.crescenzo.nom.fr/`.

[5] G. BOOCH, I. JACOBSON, et J. RUMBAUGH. *The Unified Modeling Language User Guide*. The Object Technology Series. Addison-Wesley Publishing Co., octobre 1998.

[6] M. N. BOURAQADI SAÂDANI. « *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses - Application à la programmation par aspects* ». Thèse de Doctorat, Université de Nantes, juillet 1999.

[7] R. J. BRACHMAN. « What *isa* is and isn't : an analysis of taxonomic links in semantic nets ». IEEE Computer, octobre 1983.

[8] A. CAPOUILLEZ. « ROOPS : un service de persistance paramétrable pour OFL ». Rapport Technique 99-14, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, juin 1999. Rapport de stage de DEA, `http://www.i3s.unice.fr/~acapouil/`.

[9] A. CAPOUILLEZ. « ROOPS : un service paramétrable de persistance pour OFL ». Rapport Technique 99-15, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, septembre 1999. Rapport de stage d'ESSI 3, `http://www.i3s.unice.fr/~acapouil/`.

[10] A. CAPOUILLEZ. « *La séparation des préoccupations pour l'intégration de services - Application à OFL* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, 2003. À paraître, `http://www.i3s.unice.fr/~acapouil/`.

[11] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Customization of Relationships between Types ». dans *Springer Verlag, LNCS series 1964, Posters lors d'ECOOP'2000 (14th European Conference on Object-Oriented Programming)*, juin 2000. `http://www.crescenzo.nom.fr/`.

[12] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Gestion des objets persistants grâce aux liens entre classes ». dans *OCM'2000 (Objets, Composants, Modèles « Passé, Présent, Futur »)*, mai 2000. également Rapport de Recherche I3S/RR–2000-02–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[13] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « How to Improve Persistent Object Management using Relationship Information ? ». dans *WOON'2000 (4^{th} International Conference The White Object Oriented Nights)*, juin 2000. également Rapport de Recherche I3S/RR–2000-01–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[14] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Modeling Hypergeneric Relationships between Types in XML ». dans *Scientific Bulletin of the Politehnica University of Timisoara, Symposium Etc'2000 (4^{th} Edition of Symposium of Electronics and Telecommunications)*, novembre 2000. également article pour le Workshop *Reflection and Metalevel Architectures* dans la Conférence ECOOP'2000 (14^{th} European Conference on Object-Oriented Programming) et Rapport de Recherche I3S/RR–2000-05–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[15] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Towards a More Suitable Class Hierarchy for Persistent Object Management ». dans *Springer Verlag, LNCS series 1964, ECOOP'2000 (Workshop Objects and Classification : a Natural Convergence)*, juin 2000. également Rapport de Recherche I3S/RR–2000-07–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[16] A. CAPOUILLEZ, R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Hypergénéricité pour les langages à objets : le modèle OFL ». dans *LMO"2001 (Langages et Modèles à Objets)*. Hermes Science Publications, *L'objet : logiciels, bases de données, réseaux*, volume 7, numéro 1-2/2001, janvier 2001. également Rapport de Recherche I3S/RR–2000-10–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[17] A. CAPOUILLEZ, P. CRESCENZO, et P. LAHIRE. « Separation of Concerns in OFL ». dans *ECOOP'2001 (Workshop Advanced Separation of Concerns)*, juin 2001. également Rapport de Recherche I3S/RR–2001-07–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[18] A. CAPOUILLEZ, P. CRESCENZO, et P. LAHIRE. « Le modèle OFL au service du méta-programmeur - Application à Java ». dans *LMO'2002 (Langages et Modèles à Objets)*. Hermes Science Publications, *L'objet : logiciels, bases de données, réseaux*, volume 8, numéro 1-2/2002, janvier 2002. également Rapport de Recherche I3S/RR–2001-04–FR (Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis), `http://www.crescenzo.nom.fr/`.

[19] D. CAROMEL, W. KLAUSER, et J. VAYSSIERE. « Towards Seamless Computing and Metacomputing in Java ». dans Geoffrey C. FOX, Direction de la Publication, *Concurrency Practice and Experience*, pages 1043–1061. Wiley & Sons, Ltd., septembre-octobre 1998. `http://www.inria.fr/oasis/ProActive/`.

[20] R. G. G. CATTELL, D. K. BARRY, D. BARTELS, M. BERLER, J. EASTMAN, S. GAMERMAN, D. JORDAN, A. SPRINGER, H. STRICKLAND, et D. WADE. *Object Database Standard : ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[21] P. CHAN. *Le dictionnaire officiel Java 2*. Eyrolles, mai 1999.

[22] S. CHIBA. « A Metaobject Protocol for C++ ». dans *OOPSLA'1995*, pages 285–299, octobre 1995. `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`.

[23] S. CHIBA. « OpenC++ Programmer's Guide for Version 2 ». Rapport Technique SPL-96-024, Xerox PARC, 1996. `http://www.csg.is.titech.ac.jp/~chiba/openc++.html`.

[24] S. CHIBA. « Javassist - A Reflection-based Programming Wizard for Java ». dans *OOPSLA'98 (Workshop on Reflective Programming in C++ and Java)*, octobre 1998. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`.

[25] S. CHIBA et M. TATSUBORI. « Yet Another java.lang.Class ». dans *ECOOP'1998 (Workshop on Reflective Object-Oriented Programming and Systems)*, juillet 1998. `http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/`.

[26] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « An Extensible Environment for Views in Eiffel ». Rapport Technique 96-53, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, novembre 1996. `http://www.crescenzo.nom.fr/`.

[27] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Liens entre classes dans les langages à objets ». Rapport Technique 97-22, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, décembre 1997. `http://www.crescenzo.nom.fr/`.

[28] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « An Open Object Model based on Class and Link Semantics Customization ». Rapport Technique 99-08, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, mars 1999. `http://www.crescenzo.nom.fr/`.

[29] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Customization of Links between Classes ». Rapport Technique 99-18, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, novembre 1999. `http://www.crescenzo.nom.fr/`.

[30] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « OFL : une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles ». Rapport Technique 99-09, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, mars 1999. `http://www.crescenzo.nom.fr/`.

[31] R. CHIGNOLI, P. CRESCENZO, et P. LAHIRE. « Un modèle de paramétrage des liens entre classes ». Rapport Technique 99-13, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, août 1999. `http://www.crescenzo.nom.fr/`.

[32] G. CLAVEL, N. MIROUZE, E. PICHON, et M. SOUKAL. *Java : la synthèse*. Informatique. InterEditions, 3e édition, octobre 2000.

[33] P. COINTE. « The ClassTalk System : a Laboratory to Study Reflection in Smalltalk ». dans *OOPSLA/ECOOP'1990 (First Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programmming)*, octobre 1990.

[34] P. COINTE et J.-P. BRIOT. « ClassTalk : une transposition des métaclasses d'Objvlisp à Smalltalk-80 ». dans *RFIA'1989*, novembre-décembre 1989.

[35] P. COLLET. « *Un modèle fondé sur les assertions pour le génie logiciel - Application au langage OQUAL, une extension d'Eiffel* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, octobre 1997. http://deptinfo.unice.fr/~collet/.

[36] P. COLLET et R. ROUSSEAU. « Assertions are Objects too ! ». dans *WOON'1996 (International Conference The White Object Oriented Nights)*, juin 1996. http://deptinfo.unice.fr/~collet/.

[37] S. CRAWLEY, S. DAVIS, J. INDULSKA, S. MCBRIDE, et K. RAYMOND. « Meta-meta is better-better ! ». dans *DAIS'1997 (Distributed Applications and Interoperable Systems)*, septembre-octobre 1997. http://www.dstc.edu.au/Research/Projects/MOF/[Publications.html.

[38] P. CRESCENZO. « Un système de vues pour Eiffel ». Rapport de stage de DEA, Université de Nice-Sophia Antipolis, juin 1995. http://www.crescenzo.nom.fr/.

[39] P. CRESCENZO. « OFL : les relations et descriptions d'Eiffel et de Java ». Rapport Technique I3S/RR–2001-06–FR, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, avril 2001. http://www.crescenzo.nom.fr/.

[40] P. CRESCENZO. « *OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, décembre 2001. *Cette* autoréférence*, incluse par erreur dans le mémoire initial, est conservée ici en souvenir de l'éclat de rire qu'elle provoqua dans le jury de cette thèse.*, http://www.crescenzo.nom.fr/.

[41] P. CRESCENZO. « The OFL Model to Customise Operational Semantics of Object-Oriented Languages - Application to Inter-Classes Relationships ». dans *ECOOP'2001 (11^{th} PhD. Workshop in Object-Oriented Systems)*, juin 2001. http://www.crescenzo.nom.fr/.

[42] P. CRESCENZO. « Supports de Cours d'*Informatique Appliquée au Traitement des Données* ». Conservatoire National des Arts et Métiers, depuis 1999, dernière version en 2001. http://www.crescenzo.nom.fr/.

[43] P. CRESCENZO. « Supports de Travaux Pratiques d'*Informatique Appliquée au Traitement des Données* ». Conservatoire National des Arts et Métiers, depuis 1999, dernière version en 2001. http://www.crescenzo.nom.fr/.

[44] P. CRESCENZO. « Supports de Cours de *Systèmes de Gestion de Bases de Données* ». Conservatoire National des Arts et Métiers, depuis 2000, dernière version en 2002. http://www.crescenzo.nom.fr/.

[45] P. CRESCENZO. « Supports de Cours de *Bases de Données* ». Conservatoire National des Arts et Métiers, depuis 2001, dernière version en 2002. http://www.crescenzo.nom.fr/.

[46] P. CRESCENZO, E. DARQUE-CERETTI, P. MARCENAC, et J.-L. WYBO. « METAN : a Learning System for Training of Engineers to Material Observation and Analysis ». dans *CALISCE'94 (Computer Aided Learning and Instruction in Science and Engineering)*, août-septembre 1994. http://www.crescenzo.nom.fr/.

[47] P. CRESCENZO et J.-L. PAQUELIN. « Inférence de type et langages à objets à classes ». Rapport Technique I3S/RR–2000-03–FR, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis, février 2000. `http://www.crescenzo.nom.fr/`.

[48] P. CRESCENZO et J.-L. PAQUELIN. « Type Inference in Object-Oriented Languages with Classes for Linguistic Engineering ». Séminaire invité dans le cadre d'un programme de coopération scientifique européen (Action intégrée franco-allemande : programme PROCOPE), mars 2000. `http://www.crescenzo.nom.fr/`.

[49] C. DELANNOY. *La référence du C norme ANSI/ISO*. Eyrolles, 1998.

[50] P. DESFRAY. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[51] J. DOWLING, T. SCHÄFER, V. CAHILL, P. HARASZTI, et B. REDMOND. « Using Reflection to Support Dynamic Adaptation of System Software : A Case Study Driven Evaluation ». dans *OORASE/OOPSLA'1999 (Workshop on Reflection and Software Engineering)*, 1999. `http://www.dsg.cs.tcd.ie/~coyote/`.

[52] S. DUCASSE. « *Intégration réflexive des dépendances dans un modèle à classes* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, janvier 1997.

[53] S. DUCASSE, M. BLAY-FORNARINO, et A.-M. PINNA-DERY. « A Reflective Model for First Class Relationships ». dans *OOPSLA'1995*, 1995.

[54] R. DUCOURNAU, J. EUZENAT, G. MASINI, et A. NAPOLI. *Langages et modèles à objets : état des recherches et perspectives*. Collection didactique. INRIA, juillet 1997.

[55] D. FLANAGAN. *Java in a Nutshell : a Desktop Quick Reference*. O'Reilly, 3e édition, décembre 1999.

[56] E. GAMMA, R. HELM, R. JOHNSON, et J. VLISSIDES. *Design Patterns : Catalogue de modèles de conception réutilisables*. Addison-Wesley Publishing Co., 1999.

[57] A. GOLDBERG. *Smalltalk-80 : The Interactive Programming Environment*. Computer Science. Addison-Wesley Publishing Co., 1984.

[58] A. GOLDBERG et D. ROBSON. *Smalltalk-80 : The Language and its Implementation*. Computer Science. Addison-Wesley Publishing Co., 1983.

[59] M. GOLM et J. KLEINÖDER. « metaXa and the Future of Reflection ». dans *OOPSLA'1998 (Workshop on Reflective Programming in C++ and Java)*, octobre 1998. `http://www4.informatik.uni-erlangen.de/Projects/PM/Java/`.

[60] M. GOLM, J. KLEINÖDER, et F. BELLOSA. « Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System ». dans *HotOS-VIII Workshop on Hot Topics in Operating Systems*, mai 2001. `http://www4.informatik.uni-erlangen.de/Projects/JX/`.

[61] J. GOSLING, B. JOY, G. STEELE, et G. BRACHA. *The Java Language Specification*. The Sun Microsystems Press Java Series. Sun Microsystems, juin 2000. `http://java.sun.com/docs/books/jls/`.

[62] B. GOWING et V. CAHILL. « Meta-object Protocols for C++ : The Iguana Approach ». dans *Reflection'1996*, 1996. http://www.dsg.cs.tcd.ie/~coyote/.

[63] B. HABERT. *Objectif : CLOS - Guide d'autoformation en dix séquences*. Masson, 1996.

[64] C. S. HORSTMANN et G. CORNELL. *Au cœur de Java 2 : fonctions avancées*, volume 2. Sun Microsystems, juillet 2000.

[65] C. S. HORSTMANN et G. CORNELL. *Au cœur de Java 2 : notions fondamentales*, volume 1. Sun Microsystems, juillet 2000.

[66] W. HÜRSCH et C. V. LOPES. « Separation of Concerns ». Rapport Technique NU-CCS-95-03, Northeastern University, février 1995.

[67] I. JACOBSON, G. BOOCH, et J. RUMBAUGH. *Unified Software Development Process*. The Object Technology Series. Addison-Wesley Publishing Co., janvier 1999.

[68] S. KEENE. *Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS*. Addison-Wesley Publishing Co., 1989.

[69] B. KERNIGHAN et D. RITCHIE. *Le langage C Norme ANSI*. Masson, 2e édition, 1997.

[70] G. KICZALES, J. DES RIVIÈRES, et D. G. BOBROW. *The Art of the MetaObject Protocol*. MIT-Press, 1991.

[71] J. KLEINÖDER et M. GOLM. « MetaJava : An Efficient Run-Time Meta Architecture for Java ». dans *IWOOOS'1996 (International Workshop on Object Orientation in Operating Systems)*, octobre 1996. http://www4.informatik.uni-erlangen.de/Projects/PM/Java/.

[72] P. LAHIRE. « *Conception et réalisation d'un modèle de persistance pour le langage Eiffel* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, mai 1992. http://www-iutinfo.unice.fr/~lahire/.

[73] M. LAI. *UML : la notation de modélisation objet - Applications en Java*. InterEditions (Masson), 1997.

[74] J. MALENFANT. « *Abstraction, encapsulation et réflexion dans les langages à prototypes* ». Thèse d'Habilitation à Diriger les Recherches, Université de Nantes, 1997.

[75] G. MASINI, A. NAPOLI, D. COLNET, D. LÉONARD, et K. TOMBRE. *Les langages à objets : langages de classes, langages de frames, langages d'acteurs*. MIT Press, 1991.

[76] B. MEYER. « Applying "Design by contract" ». *IEEE Computer*, 25(10), octobre 1992.

[77] B. MEYER. *Eiffel : The Language*. Object-Oriented Series. Prentice Hall, 1992. http://www.eiffel.com/doc/.

[78] B. MEYER. *Eiffel, le langage*. InterEditions, 1994. http://www.eiffel.com/doc/documentation.html#etl (English Version).

[79] B. MEYER. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2e édition, 1997. http://www.eiffel.com/doc/oosc/.

[80] A. MICHARD. *XML Language and Applications*. Eyrolles, 1999.

[81] OBJECT MANAGEMENT GROUP. « *XML MetaData Interchange (XMI) Specification* », novembre 2000. Version 1.1, `http://www.omg.org/technology/xml/`.

[82] OBJECT MANAGEMENT GROUP. « *Meta Object Facility Specification (MOF)* », novembre 2001. Version 1.3.1, `http://www.omg.org/technology/documents/formal/meta.htm`.

[83] OBJECT MANAGEMENT GROUP. « *Unified Modeling Language Specification (UML)* », février 2001. Version 1.4, `http://www.omg.org/technology/uml/`.

[84] A. OLIVA. « *The Guaraná API* ». *Institute of Computing of the State University of Campinas*, 2001. `http://sunsite.unicamp.br/~oliva/guarana/`.

[85] M. C. OUSSALAH, Direction de la Publication. *Génie objet : analyse et conception de l'évolution*. Hermes, septembre 1999.

[86] M. C. OUSSALAH et *alii*. *Ingénierie objet : concepts et techniques*. Informatiques. InterEditions, mai 1997.

[87] D. PESCARU et P. LAHIRE. « OFL-ML : A Modelling Language for Handling Customisation of Classes and Relationships ». Rapport de Recherche, août 2001.

[88] POET SOFTWARE. « Site Web de *FastObjects* ». World Wild Web, 2002. `http://www.fastobjects.com/`.

[89] F. RIVARD. « *Évolution du comportement des objets dans les langages à classes réflexifs* ». Thèse de Doctorat, Université de Nantes, 1997.

[90] J. RUMBAUGH, I. JACOBSON, et G. BOOCH. *The Unified Modeling Language Reference Manual*. The Object Technology Series. Addison-Wesley Publishing Co., décembre 1998.

[91] B. STROUSTRUP. *The Design and Evolution of C++*. Addison-Wesley Publishing Co., 2^{e} édition, janvier 1994.

[92] B. STROUSTRUP. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3^{e} édition, 1997. `http://www.research.att.com/~bs/3rd.html`.

[93] SUN MICROSYSTEMS, INC.. « Java Remote Method Invocation - Distributed Computing for Java ». `http://java.sun.com/j2se/1.3/docs/guide/rmi/`, novembre 1999.

[94] SUN MICROSYSTEMS, INC.. « Java Remote Method Invocation Specification ». `http://java.sun.com/products/jdk/rmi/`, décembre 1999.

[95] R. SWITZER. *Eiffel : An Introduction*. Prentice Hall, 1993.

[96] G. TALENS. « *Gestion des objets simples et composites* ». Thèse de Doctorat, Université de Montpellier II, février 1994.

[97] M. TATSUBORI, S. CHIBA, M.-O. KILLIJIAN, et K. ITANO. « OpenJava : A Class-Based Macro System for Java ». dans *Springer Verlag, LNCS series 1826, Reflection and Software Engineering*, pages 117–133, 2000. `http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/`.

[98] M. TATSUBORI, T. SASAKI, S. CHIBA, et K. ITANO. « A Bytecode Translator for Distributed Execution of "Legacy" Java Software ». dans *ECOOP'2001 (Springer Verlag, LNCS series 2072)*, pages 236–255, 2001. http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[99] P. THOMAS et R. WEEDON. *Object-Oriented Programming in Eiffel*. Addison Wesley, 1995.

[100] I. WELCH et R. STROUD. « Dalang - A Reflective Extension for Java ». Rapport Technique CS-TR-672, *Computing Science Department, University of Newcastle upon Tyn*, septembre 1999. http://www.cs.ncl.ac.uk/people/i.s.welch/home.formal/-dalang/.

[101] I. WELCH et R. STROUD. « *Lecture Notes in Computer Science 1826* », Chapitre Kava - A Reflective Java based on Bytecode Rewriting. Springer Verlag, 2000. http://www.cs.ncl.ac.uk/research/dependability/reflection/.

[102] World Wide Web Consortium. « *Extensible Markup Language (XML)* », 2e édition, octobre 2000. Version 1.0, W3C Recommendation, http://www.w3.org/TR/REC-xml.

Résumé

Le modèle *OFL* (*Open Flexible Languages*) décrit et réifie le comportement des langages à objets à classes courants tels *Java*, *C++* ou *Eiffel* et permet de modifier, d'adapter, ce comportement. Dans cette optique, les notions de classe — généralisée sous le nom de *description* — et de *relation entre descriptions* sont réifiées.

Chaque langage est décrit sous la forme d'un ensemble de composants formé de types de description et de types de relation. Pour définir chaque composant, le méta-programmeur doit donner une valeur à un ensemble de paramètres qui décrivent la sémantique opérationnelle du composant. Par exemple, pour définir une nouvelle relation, il devra répondre aux questions :

- La relation définit-elle une utilisation ou une importation ?
- La relation permet-elle le polymorphisme ?
- Si oui, dans quel(s) sens ?
- Quelle est sa cardinalité maximale ?
- Peut-elle être circulaire, répétée ?
- Possède-t-elle une relation inverse ?
- ...

Une fois ces types de description et de relation définis par le méta-programmeur, le programmeur est en mesure d'en faire usage pour la réalisation de son application.

Le but du modèle *OFL* est d'aider à améliorer la qualité du code produit en donnant plus d'information et donc de précision sur les relations entre descriptions. Nous souhaitons de cette manière favoriser l'évolution et la maintenabilité des applications par l'intermédiaire d'une meilleure spécification, d'une documentation plus explicite, de contrôles automatiques plus pertinents, ... Il s'agit également d'offrir une plus grande souplesse au programmeur en lui permettant, par une phase de paramétrage ou de méta-programmation, de mieux adapter son langage de programmation de prédilection à ses besoins.

L'objectif de cette thèse est de définir le modèle *OFL* et d'en présenter des exemples d'application.

Discipline

Informatique

Mots-clés

langages à objets, protocole méta-objet, paramétrage hyper-générique, relations inter-classes

Laboratoire d'accueil

Laboratoire I3S (Informatique, Signaux et Systèmes de Sophia-Antipolis)
UNSA/CNRS
Projet OCL
2000, route des lucioles
Les Algorithmes, Bâtiment Euclide B
BP 121
F-06903 Sophia Antipolis CEDEX
France